REINFORCEMENT LEARNING WITH TIME PERCEPTION

A thesis submitted to the University of Manchester
for the degree of Doctor of Philosophy
in the Faculty of Engineering and Physical Sciences

2012

By
Chong Liu
School of Computer Science

Contents

Word Count: 93,166

List of Tables

List of Figures

Abstract

Classical value estimation reinforcement learning algorithms do not perform very well in dynamic environments. On the other hand, the reinforcement learning of animals is quite flexible: they can adapt to dynamic environments very quickly and deal with noisy inputs very effectively. One feature that may contribute to animals' good performance in dynamic environments is that they learn and perceive the time to reward.

In this research, we attempt to learn and perceive the time to reward and explore situations where the learned time information can be used to improve the performance of the learning agent in dynamic environments. The type of dynamic environments that we are interested in is that type of switching environment which stays the same for a long time, then changes abruptly, and then holds for a long time before another change. The type of dynamics that we mainly focus on is the time to reward, though we also extend the ideas to learning and perceiving other criteria of optimality, e.g. the discounted return, so that they can still work even when the amount of reward may also change.

Specifically, both the mean and variance of the time to reward are learned and then used to detect changes in the environment and to decide whether the agent should give up a suboptimal action. When a change in the environment is detected, the learning agent responds specifically to the change in order to recover quickly from it. When it is found that the current action is still worse than the optimal one, the agent gives up this time's exploration of the action and then remakes its decision in order to avoid longer than necessary exploration.

The results of our experiments using two real-world problems show that they have effectively sped up learning, reduced the time taken to recover from environmental changes, and improved the performance of the agent after the learning converges in most of the test cases compared with classical value estimation reinforcement learning algorithms. In addition, we have successfully used spiking

neurons to implement various phenomena of classical conditioning, the simplest form of animal reinforcement learning in dynamic environments, and also pointed out a possible implementation of instrumental conditioning and general reinforcement learning using similar models.

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright

Acknowledgements

There are many people to whom I am indebted during my PhD research. First of all, I would like to thank my supervisor Dr. Jonathan Shapiro and advisor Professor Steve Furber for their continuous help and guidance.

Another great debt is to other lecturers and my fellow PhD students of the school as well as researchers outside the school or the university. I really enjoy both casual conversation and academic discussion with them. In particular, I thank Joy Bose and Dr. Stefano Panzeri for introducing me to spiking neurons, John M. Butterworth for his discussion of reinforcement learning with me, Dr. Michael L. Anderson for answering my questions about the MCL approach, and Dr. Jeremy Wyatt and Dr. Xiaojun Zeng for their suggestions for improving this thesis. In addition, I am also very grateful to Geoff and Sylvia, the former organisers of Globe Cafe Manchester, through which I have learned a lot about British culture and also made many good friends. In particular, I thank Vicky and Mark for their efforts to help me correct all the grammar errors in this thesis.

I gratefully acknowledge the financial support of this research from the Overseas Scholarship Scheme provided by Universities UK and from the departmental studentship provided by the School of Computer Science at the University of Manchester.

Finally, thanks go to my grandmas, parents, parents-in-law, brothers and sisters for their support, help and encouragement. I especially thank my angel, Tian. Without her support and encouragement, I would never have finished this work. I am thankful for our unborn baby whose arrival will be a joy and also an inspiration to me and whose birth will be a big blessing to our whole family.

Chapter 1

Introduction

Reinforcement learning (RL) [1, 2] is learning from interaction with a dynamic environment and from the consequences of the learning agent's own actions, rather than from explicit teaching. It is intended to address one prediction, learning and decision making problem that animals and humans have to face in their everyday lives. In addition, it has found a great number of applications in the real world, such as game playing, robotics and industrial manufacturing.

This research attempts to learn and monitor the time to reward and explore situations where the learned time information can be used to improve the performance of the learning agent in dynamic environments. The motivation for our research is discussed in section 1.1. Section 1.2 presents our research aims. Section 1.3 explains the meaning of time perception in the context of this research. In section 1.4, the novelty of our research is discussed. The last section outlines the whole thesis.

1.1 Research motivation

Classical value estimation reinforcement learning algorithms, e.g. Monte Carlo methods [3–5], dynamic programming algorithms [6], Q learning [7], SARSA learning [8], and Prioritized Sweeping [9], only learn the value of reward but use a discount factor to distinguish two reward with the same value but received at different times (a reward received in the future would become smaller after the discount). In effect, they use a single value for two things, viz. the value of reward and the time to reward.

This method, however, suffers from several drawbacks. Firstly, it cannot

distinguish a small reward r_1 received in a shorter time t_1 from a big reward r_2 received in a longer time t_2 as long as $r_1\gamma^{t_1} = r_2\gamma^{t_2}$ where $0 \leq \gamma \leq 1$ is the discount factor. Although this is probably a favourable behaviour in general cases, it may not be desirable in some restrictive cases. For instance, a robot with limited battery tries to find a charging point (reward). Suppose the battery is enough to support the robot for t_1 but not for t_2. The optimal decision in this case should go for the first charging point rather than the second one. In addition, it cannot distinguish a reward with constant value or constant time to reward and the other one with variable value or variable time to reward as long as the mean of the discounted values of the two rewards is the same. Animal experiments [10, 11], however, showed that even when the expected value of the rewards is the same, some animals may be risk-prone (choose the one with variable value or variable time to reward) [12,13], while others are risk-averse (choose the one with constant value or constant time to reward) [14, 15], or sometimes indifferent to risk (the variability of the value or the variability of the time to reward) [16,17]. Furthermore, some studies [18,19] also suggested that the preference depends on the energy budget of the animals.

Furthermore, although continuous exploration is needed in nonstationary environments, it is not necessary to continue this time's exploration of the current action or the current state-action pair beyond the time when the learning agent has found that the action or the state-action pair is still worse than the optimal one. It is worth noting, however, that this does not affect future exploration of the action or the state-action pair. For simplicity, we consider a deterministic environment with only one state and two actions. Suppose that the value of reward for both actions is the same and does not change but the time to reward for both actions may change. Suppose that the agent has learned that it takes 1 minute to get a reward if it chooses the first action or 1 year to get the same amount of reward if it selects the second action. Even though the first action is apparently the optimal action at present, the agent still needs to explore/select the second action occasionally just in case the environment has changed and the time to reward for the second action has become even shorter than that for the first action, e.g. 1 second. This is the classical trade-off between exploration and exploitation. On the other hand, however, does the agent need to wait (1 year) until receiving a reward after it takes the second action? Fortunately, this is not

necessary. If the reward has not arrived 1 minute after the second action is selected, the agent can conclude that the second action is still worse than the first one, so the purpose of this time's exploration has been fulfilled and therefore the agent does not need to wait any longer. By giving up the second action 1 minute after it is taken and then choosing the first action instead, the agent can get a reward in only 2 minutes in this episode, whereas it may take the agent 1 year to get a reward if it chooses to continue its wait until receiving a reward after it takes the second action. Even if the time to reward is not deterministic or the amount of reward for actions in one state is not the same, it is still possible to find a time after which the current action is still worse than the optimal one and therefore longer exploration beyond the time is not necessary. Unfortunately, however, the learned discounted reward alone cannot be used to decide when the agent should give up this time's exploration of a suboptimal action because it may expect a bigger reward in a longer time which would have the same value of discounted reward. In contrast, animal experiments [20–22] showed that animals maintain the average foraging rate. When the foraging rate in the current patch is below the average one, they give up the patch and then choose other patches instead.

Similarly, if the agent knows the expected time to a reward after taking an action, it can deduce that something wrong must have happened, e.g. the environment has changed, it has made a bad decision or it has taken a different action from what it had intended to take, when it has not received the reward long after the expected time. Being able to discover that something is wrong enables the agent to change its policy accordingly and therefore can potentially improve its performance. For example, suppose that it usually takes a person 30 minutes to drive home from work. One day, however, he has not yet arrived home after more than one hour's driving or he finds himself in a completely unfamiliar place. If everything is as usual, e.g. traffic free-flowing, weather good and no road closure, he would realise that he had made a wrong turn at one junction. If he goes back to the junction or some familiar place, it will not take him very long to reach his destination if he does not make another wrong turn afterwards. On the other hand, however, if he continues his journey, he may be further away from his home and may never even arrive. Animal experiments also found that animals have the ability to learn and perceive the time to reward [23–25] and can also

use this ability to detect if something is wrong and then adjust its policy accordingly [10]. Specifically, the experiment by Brunner et al. [26] on starlings showed that the rate of their pecking (active search of foods) peaked when the time was close to the time of the reward; the starlings stopped pecking about 1.5 times the inter-prey interval if they had not received any reward, and then abandoned the current patch and flew to other patches 13s later. In addition, the hungry feeling of animals also serves a similar purpose. When animals feel hungry, they know that their energy level is low and they need to eat something in order to top up its energy. In contrast, if animals had no feeling of hunger, they would not be able to know that their energy is low and would have no incentive to top up its energy. Instead, they may continue consuming their energy and would eventually run out of energy and die.

Next, in a dynamic environment, although these classical value estimation algorithms can forget past experience (weight recent rewards more heavily) through a fixed learning rate α, keep exploring through ϵ-greedy or other non-greedy decision making strategies, and change their policies by updating their value functions when the environment changes, the learning process is usually quite slow after the environment changes because it takes the learning agent a long time to unlearn the previous optimal policy [27]. Thus, it may take them a long time to recover from an environmental change. Experiments by Anderson et al. [28, 29] further showed that in some cases it is even better to throw away the existing policy and start over than to continue learning with the existing policy. In addition, in order to keep tracking nonstationary environments, the learning rate cannot be reduced to a very small number. For this reason, the learned value cannot converge to its true value even in a stationary but stochastic environment. If there is a method that can detect changes in the environment, the learning rate can be increased to learn the change quickly when an environmental change has been detected and can be decreased for the estimated value to converge to its true value when an environmental change has not been detected. Studies in neuroscience [30,31] found that animals can detect environmental changes, and then boost their learning rate when an environmental change is detected, and reduce their learning rate otherwise. Specifically, when the consequence of a stimulus is uncertain and the uncertainty is expected (e.g. stochastic environments), the activity of neocortical acetylcholine (ACh) increases [32]; when the consequence of a stimulus is not expected (e.g. nonstationary environments), the activity of

neocortical norepinephrine (NE) increases [33]. Although ACh reports on uncertainties in internal estimates whereas NE reports on dramatic changes, the increased level of both ACh and NE will increase the learning rate, boost learning about the environment and enhance bottom-up processing in inference [30, 31].

For these reasons, we attempt to incorporate some of the above features of animal learning into reinforcement learning in order to improve the performance of the learning agent in dynamic environments. Specifically, we attempt to learn and monitor the time to reward and then use it to detect changes in the environment. When a change is detected, the learning agent responds specifically to it in order to recover from it quickly. In addition, the learned time information is also used to find out when the agent should give up the current action in order to avoid longer than necessary exploration. Our long term research plan is to implement reinforcement learning together with the above features of animal learning using biologically plausible neuron models which are the foundations of animal learning and behaviour.

1.2 Research aims

This PhD research aims to learn and perceive the time to reward and explore situations where the learned time information can be used to improve the performance of the learning agent in dynamic environments. The type of dynamic environments that we are interested in is that type of switching environment which stays the same for a long time, then changes abruptly, and then holds for a long time before another change. The type of dynamics that we mainly focus on is the time to reward.

In addition, as part of ongoing research, which attempts to build a biologically plausible neuron model that is capable of implementing reinforcement learning and of adapting to dynamic environments quickly, this PhD research also explores the possibilities of using biologically plausible neuron models to implement reinforcement learning.

1.3 What is time perception?

The definition of time perception from Encyclopædia Britannica [34] is

> "experience or awareness of the passage of time".

For animals, the ability to perceive time serves at least four purposes. Firstly, they learn the timing of an event if the event is regular. The event may be the arrival of prey or the arrival of winter. Secondly, they use the learned time to predict the approach of the event so that they can prepare for it in advance (e.g. in classical conditioning, the salivation of dogs prepares them for digesting food). Thirdly, they compare the learned timing of the event with the actual timing of the event to find out if the timing of the event has changed. If it has changed, they respond quickly to the change. In Pavlov's experiments, for instance, if both the timing when the bell sounds and the timing when the food is given are delayed, dogs will also delay their salivation. Finally, if the event does not happen some time after the predicted time, they know that the event will not come or the action choice they made is not a good choice and therefore need not wait for it forever. Therefore, in this research, the term *time perception* not only means perceiving time but also means the behaviours associated with it, viz. learning, predicting, monitoring and comparing time.

1.4 What is novel in this research?

1. We have successfully implemented various phenomena of classical conditioning using spiking neurons [35] and also pointed out a possible implementation of instrumental conditioning and general reinforcement learning using similar models. Traditionally, classical conditioning is mainly modelled by connectionist neural networks, e.g. the Rescorla-Wagner model [36] and the Sutton-Barto model [37]. These models use high-level abstractions of neurons which ignore the temporal dynamics of real neurons. Classical conditioning, however, is dynamic in essence because it mainly learns and predicts the dynamic environment. The Sutton-Barto model solves this problem by using a temporal learning rule. Here we attempt to use the temporal dynamics of biological neurons directly to model the dynamic nature of classical conditioning.

2. We have designed a method that detects changes in the environment by comparing the actual value of a state-action pair (s, a) in the current trial with the estimated mean $(Q(s, a))$ and variance $(Q\_var(s, a))$ of the value of (s, a). When the actual value of (s, a) is outside $Q(s, a) \pm k\sqrt{Q\_var(s, a)}$ where $k \geq 0$, the method deems that the value of (s, a) has changed. This

method has the potential to detect changes in the environment with only one trial even in a stochastic environment. On the other hand, if only the mean of the value is learned, many trials are needed to detect changes in a stochastic environment because it has to compare the mean of the values in recent several trials with the mean of the values in several trials before recent several trials. Furthermore, this method is much easier than to learn the full distribution of the values, which is a non-trivial task in its own right.

3. We have designed a method to adjust the value of the learning rate to different situations. The learning rate is increased to learn the change quickly when an environmental change has been detected and the learning rate is decreased towards 0 for the estimated value to converge to its true value when an environmental change has not been detected. Traditionally, a fixed learning rate is usually used. The value of the fixed learning rate cannot be either too small or too big because it has to balance the nonstationary stages of the environment and the stationary stages of the environment. When the environment changes, the learning rate is not big enough to respond quickly to the change. On the other hand, when the environment does not change, the learning rate is not small enough for the estimated value to converge to its true value if the environment is stochastic.

4. We have used two methods to balance exploration and exploitation. Firstly, the exploration rate (e.g. ϵ for ϵ-greedy) is increased in order to increase the chance that suboptimal actions are visited if it is found that a suboptimal action in one state has improved and may potentially become the optimal action; otherwise, the exploration rate is decreased gradually towards its minimum value in order to reduce the cost of exploring suboptimal actions. Secondly, if the learning agent has found that the current action in one state is still worse than the optimal one in the state some time after taking the action, it will give up this time's exploration of the action in order to avoid longer than necessary exploration. It is worth mentioning that it is impossible to completely avoid exploration in nonstationary environments because only when the learning agent has explored the state-action pairs can it know whether their values have changed or not. However, with our methods, the cost of exploration can be dramatically reduced in certain

cases.

5. The learning rate is associated with each state-action pair. Traditionally, a single learning rate is used for all state-action pairs. When the environment changes, it makes sense to increase the learning rate in order to respond quickly to the environmental change. However, not all state-action pairs may have been affected by the change. It may lead to instability if the learning rates of the state-action pairs, whose values have not changed, are increased in a stochastic environment. Therefore, we use a different learning rate for each state-action pair so that only the learning rates of the pairs, whose values have changed, are increased when the environment changes.

6. The exploration rate (e.g. ϵ for ϵ-greedy) is associated with each state. Traditionally, a single ϵ is used for all states. When the environment changes, ϵ is usually increased to explore suboptimal actions. However, only when the suboptimal actions have improved does increasing ϵ help. In some cases, for example, when the optimal action has improved or the suboptimal actions have worsened, the result is even worse if ϵ is increased. In addition, some states may satisfy the condition of increasing ϵ whereas others are not. Therefore, we use a different ϵ for each state so that only the ϵ of the states satisfying the conditions is increased when the environment changes.

1.5 Thesis outline

A brief introduction to this research has been provided in this chapter. The remainder of this thesis is organised as follows:

Chapter 2 provides some preliminary knowledge about reinforcement learning including problems that reinforcement learning addresses, challenges for reinforcement learning and classical algorithms for reinforcement learning.

Chapter 3 discusses recent research on reinforcement learning which is related with this research and compares this research with them.

Chapter 4 introduces spiking neuron models and presents our work implementing classical conditioning using spiking neurons. In addition, it points out a possible implementation of instrumental conditioning and general reinforcement learning using similar models.

Chapter 5 and the next chapter, introduce two real-world problems and related algorithms. This chapter introduces a simple problem which has only one state but multiple actions with delayed reward. In this chapter, we also investigate possible implementations of the ideas of learning and monitoring the time to reward and then design simple algorithms for this kind of reinforcement learning problem with only one state. In addition, we also extend the idea of learning and perceiving the time to reward to learning and monitoring other criteria of optimality, e.g. the discounted reward, so that it can work even when the amount of reward is not the same and may also change. Finally, we compare the standard reinforcement learning algorithms without time/value perception with the algorithms with time/value perception in various experimental scenarios.

Chapter 6 introduces a route finder problem which naturally extends the time delayed n-armed bandit problem to multiple states and extends the algorithms introduced in last chapter to work with multiple states. In addition, this chapter also compares standard Monte Carlo methods without time perception with Monte Carlo methods with time perception in various experimental scenarios.

Chapter 7 concludes the thesis with a summary of the motivation, fundamental ideas, justifications for key decisions, and the main results, contributions and limitations of this research as well as ideas for future work.

Chapter 2

Background on reinforcement learning

Reinforcement learning (RL) [1, 2] concerns a fundamental type of learning: a learning agent learns from interaction with a dynamic environment and the consequences of its own actions rather than from explicit teaching. It is intended to address the prediction, learning and decision making problems that animals and humans have to face in their everyday lives.

Reinforcement learning is also relevant to robotics. The real world is so complex that it is impossible to hard-wire controllers for robots that can cope with every situation. In addition, it is also not realistic for humans to instruct robots all the time. They must be able to learn and behave properly and independently in different situations on the basis of the rewards or penalties they have received.

This chapter is organised as follows. In section 2.1, we introduce reinforcement learning in detail and discuss its various features. Then the challenges faced by reinforcement learning are present in section 2.2. In the last section, various methods for solving reinforcement learning problems are discussed.

2.1 Elements of reinforcement learning

In the reinforcement learning paradigm, an agent, e.g. an animal or a robot, gathers information from the environment in which it is situated, and then takes an action according to the information. After that, it will receive either a reward or a penalty immediately or later from the environment for what it has done. In the future, it will modify its strategy (or policy) to behave satisfactorily or

optimally in the environment in terms of its criteria of optimality.

2.1.1 The agent

The agent in the reinforcement learning paradigm is an entity that has sensors to perceive the state of the environment, a “brain" to make decisions based on its observations, and effectors to take actions based on its decisions.

From the above definition of the agent, it is straightforward to model the agent as (M_S, π, M_A) where M_S models the sensors of the agent and is a mapping from the set of all possible states S of the environment to the set of all possible observations S' of the states of the environment made by the agent. π models the “brain" of the agent and is a mapping from the set of all possible observations S' to the set of probability distributions across all possible intended actions. It is usually called the policy of the agent, which will be discussed in detail in the next section. M_A models the effectors of the agent and is a mapping from the set of all possible intended actions A' of the agent to the set of all possible actual actions A of the agent.

It is worth noting that the observation of one state of the environment made by the agent may not be exactly the same with the actual state of the environment because the agent may not have complete and perfect perception of the state of the environment. It may cause serious problems for the agent if two quite different states of the environment are mapped to the same observation (the many-to-one relation), viz. the agent can not distinguish two states of the environment. In addition, the agent with a faulty sensor may also perceive the same state of the environment differently on different occasions (the one-to-many relation). This case, however, does not pose a big problem to the agent. It only increases the state space perceived by the agent and therefore slows down its learning. If both situations happen, the relationship between the states of the environment and the observations made by the agent will become many-to-many. This will cause the problems arisen both in the many-to-one situation and in the one-to-many situation.

Likewise, the intended action of the agent is not necessarily the same with the actual action it takes. One classic example is that a language learner may intend to pronounce one sound but has actually pronounced another sound instead (the many-to-one relation). This case, however, similar to the one-to-many relation of the sensor, does not cause great trouble for the agent because the agent can

distinguish these two situations and therefore is able to only adjust the pronunciation/action leading to a low reward whereas leaving the pronunciation/action leading to a high reward unchanged. In addition, the agent with a faulty effector may also perform different actions on different occasions even though the agent intends to perform the same action (the one-to-many relation). Its net effect is similar to that of the many-to-one relation of the sensor. In fact, they are equivalent in some cases. For instance, suppose that an agent is in one state of an environment where it will receive a reward if it moves left whereas it will receive a penalty if it moves right. The agent has a perfect sensor but has an equal probability of moving left and of moving right when it chooses to move either left or right. This situation is equivalent to where an agent has perfect effectors but cannot distinguish between a state in which it will receive a reward if it moves left and a penalty if it moves right and another state in which it will receive a penalty if it moves left and a reward if it moves right. Similarly, many-to-many relations will also appear if both situations occur, which will cause the problems arisen both in the many-to-one situation and in the one-to-many situation.

In this thesis, however, we are only concerned with the situation where the observation of one state of the environment made by the agent is exactly the same with the state of the environment (the one-to-one relation) and the intended action of the agent is exactly the same with its actual action (the one-to-one relation).

2.1.2 The policy

In general, a policy defines the agent's way of making decisions at a given time or in a certain state/situation. In the reinforcement learning paradigm, it is a mapping from perceived states of the environment to actions taken by the agent in those states. Roughly speaking, the policy of a learning agent is just what action it chooses to do in a certain state. The learning process of the agent involves improving its policy. As for stochastic policies, the policy $\pi(s, a)$ defines the probability that action $a, a \in \mathcal{A}(s)$, is chosen in state s, viz.

$$\pi(s, a) = Pr(a_c = a | s_c = s) \tag{2.1}$$

where s_c is the state in which the agent finds itself and a_c is the action that the agent will take in s_c.

2.1.3 The environment

The environment is a closed system where the agent is situated. Basically, everything outside the agent is considered as part of the environment. The goal of the agent is to behave optimally or properly in the environment in terms of its criteria of optimality. In addition, the action of the learning agent may influence the environment's state in which it finds itself. For instance, if a lion forages in a place today, it may cause fewer prey to go to the place in the near future.

The environment can be modelled as (S, M, R) where S is the set of all possible states of the environment, $M : (S, A) \rightarrow S$ is the mapping from the current state of the environment and the action of the agent to the next state of the environment, R is the set of all possible rewards offered by the environment.

2.1.4 Rewards and returns

A reward, r, is defined as what an agent can receive by taking a certain action in a particular state of the environment in which it is situated.

The *return*, R_i, is a measure of long-term rewards and is the sum of some function of the reward sequence received after step i but before step j.

$$R_i = \sum_{k=i+1}^{j} f_k(r_k). \tag{2.2}$$

The step j and the function f_k depend on the agent's criteria of optimality, which will be discussed in subsection 2.1.9.

2.1.5 Markov property and Markov processes

In general, the outcome of a discrete stochastic process $X = (X_i, i \in I)$ at the $(i+1)^{th}$ stage is dependent on the prior sequence of outcomes, viz.

$$Pr(X_{i+1} = x_{i+1} | X_i = x_i, X_{i-1} = x_{i-1}, \ldots, X_0 = x_0). \tag{2.3}$$

A discrete stochastic process is said to have the *Markov property* if the outcome of the process at any step $i+1$ where $i, i+1 \in I$ is only dependent on the outcome of the process at step i, that is, independent on all the outcomes of the process before step i. A discrete stochastic process possessing the Markov property is called a *Markov process*. If the process X mentioned above possesses

the Markov property, the conditional probability can be simplified as

$$Pr(X_{i+1} = x_{i+1}|X_i = x_i, X_{i-1} = x_{i-1}, \ldots, X_0 = x_0) = Pr(X_{i+1} = x_{i+1}|X_i = x_i). \tag{2.4}$$

It is worth noting that the Markov property relaxes the requirement for an *independent process*, where the outcome of the process at any step is independent on all the prior sequence of outcomes. If the process X mentioned above is an independent process, the conditional probability can be further simplified as

$$Pr(X_{i+1} = x_{i+1}|X_i = x_i, X_{i-1} = x_{i-1}, \ldots, X_0 = x_0) = Pr(X_{i+1} = x_{i+1}). \tag{2.5}$$

2.1.6 Markov decision processes

The *Markov decision process (MDP)* extends the Markov process by making the transition depend on the decision or the action of an agent. Suppose that an agent in one state of an environment makes a decision (takes an action), and then enters another state of the environment and receives a reward in one discrete time step. The environment is a *Markov decision process* if the probability of transition from the current state (s_i) to the next state (s_{i+1}) and receipt of the reward (r_{i+1}) depends only on the current state of the environment and the current action of the agent, not on the previous states, actions taken and rewards received, viz. the Markov property. The Markov property for MDPs can be expressed mathematically as:

$$Pr(s_{i+1}, r_{i+1}|s_i, a_i, r_i, s_{i-1}, a_{i-1}, r_{i-1}, \cdots, r_1, s_0, a_0) = Pr(s_{i+1}, r_{i+1}|s_i, a_i). \tag{2.6}$$

If the state and action spaces are finite, the Markov decision process is called a *finite Markov decision process*. MDPs have largely simplified reinforcement learning problems, and most reinforcement learning algorithms and theories assume that the environment is a finite MDP. In real-world problems, though the Markov property is rarely strictly satisfied, many problems can be approximated by MDPs when the states are carefully constructed and represented.

2.1.7 Semi-Markov decision processes

In MDPs, the time spent in any transition is the same. This assumption, however, is not the case for many real-world problems. *Semi-MDPs* (SMDPs) extend

MDPs by allowing transitions to have different duration (t). The Markov property for semi-MDPs can be expressed mathematically as:

$$P(s_{i+1}, r_{i+1}, t_{i+1} \leq t | s_i, a_i, r_i, t_i, s_{i-1}, a_{i-1}, r_{i-1}, t_{i-1}, \cdots, r_1, t_1, s_0, a_0) \\ = P(s_{i+1}, r_{i+1}, t_{i+1} \leq t | s_i, a_i) \quad (2.7)$$

where t_{i+1} is the time taken by the transition from s_i to s_{i+1} and $t \geq 0$ is a period of time. $P(s_{i+1}, r_{i+1}, t_{i+1} \leq t | s_i, a_i)$ represents the probability that the next state is s_{i+1}, the transition from s_i to s_{i+1} takes no more than t and the agent receives reward r_{i+1} when the current state is s_i and it takes action a_i. Or it can be expressed separately by

$$P(t_{i+1} \leq t | s_i, a_i, r_i, t_i, s_{i-1}, a_{i-1}, r_{i-1}, t_{i-1}, \cdots, r_1, t_1, s_0, a_0) \\ = P(t_{i+1} \leq t | s_i, a_i) \quad (2.8)$$

and

$$P(s_{i+1}, r_{i+1} | t_{i+1}, s_i, a_i, r_i, t_i, s_{i-1}, a_{i-1}, r_{i-1}, t_{i-1}, \cdots, r_1, t_1, s_0, a_0) \\ = P(s_{i+1}, r_{i+1} | t_{i+1}, s_i, a_i). \quad (2.9)$$

2.1.8 Types of tasks

In some tasks, the interaction between the agent and the environment can be broken into subsequences or episodes with each episode ending in a special terminal state. Such tasks are called *episodic tasks*. Examples of episodic tasks include playing a game, cooking a dish and travelling home after work.

On the other hand, there are tasks that cannot be broken into episodes, such as the lifelong learning of an animal or a robot. These tasks are called *continual tasks*.

2.1.9 Criteria of optimality

In general, we want the agent to behave optimally in the environment where it is situated. Depending on what kind of optimal behaviour we want the learning agent to achieve, there are mainly three criteria of optimality in the reinforcement learning paradigm. The first one is the finite-horizon model which aims to maximise the expected return in a finite number (n) of steps. If there are infinite

steps, the agent always looks ahead n steps in every step. Suppose that the agent is in the i^{th} step and the reward sequence at step k is r_k. Under this criterion, the target of the agent is to maximise

$$E(R_i) = E(\sum_{k=i+1}^{i+n} f_k(r_k)) \tag{2.10}$$

where $f_k(r) = r$. If there are only n steps, however, the agent looks ahead n steps before taking the first step, then looks ahead $n-1$ steps after taking the first step, and so on until it terminates. Here, $i < n$. Under this criterion, the target of the agent is to maximise

$$E(R_i) = E(\sum_{k=i+1}^{n} f_k(r_k)) \tag{2.11}$$

where $f_k(r) = r$.

Another is the infinite-horizon discounted model which aims to maximise the expected discounted return in the infinite future. In this way, immediate and delayed rewards are well balanced: immediate rewards are more heavily weighted than delayed rewards. Under this criterion, the target of the agent is to maximise

$$E(R_i) = E(\sum_{k=i+1}^{\infty} f_k(r_k)) \tag{2.12}$$

where $f_k(r) = \gamma^{k-i-1} r$ and γ $(0 \leq \gamma \leq 1)$ is a parameter called the *discount factor*.

The last is the average-reward model which aims to maximise the expected long-term average return. Under this criterion, the target of the agent is to maximise

$$E(R_i) = \lim_{n\to\infty} E(\frac{1}{n} \sum_{k=i+1}^{n} f_k(r_k)) \tag{2.13}$$

where $f_k(r) = r$.

In this thesis, we are mainly concerned with the infinite-horizon discounted model unless otherwise specified.

2.2 Challenges for reinforcement learning

Due to the evaluative nature of reinforcement learning, it faces several challenges. Firstly, it needs to balance exploration and exploitation. Secondly, rewards may be received from the environment immediately or with some delay. Should they be treated equally? Thirdly, a reward may have been achieved by a sequence of actions. How should reinforcement learning algorithms assign the credit for the reward to these actions (the temporal credit assignment problem)? Fourthly, the behaviour of the agent is guided by the location and value of rewards. How should we assign different values of rewards to different locations in order for the agent to behave as we expect? Fifthly, a simple greedy policy choosing the action which leads to the maximum immediate reward does not guarantee optimality. Finally, for large problems, generalisation may be needed to estimate the value of state-action pairs that have not been experienced previously. The challenges for reinforcement learning posed by dynamic environments, however, will not be discussed here but in section 3.5 instead.

2.2.1 Evaluative vs. instructive

In the supervised learning paradigm, the learning process is *instructive*. After the learning agent takes some action according to its inputs (e.g. the current state of the environment), a teacher will inform it of the right action that it should have taken. Then, it adjusts its internal parameters to minimise the errors between its action and the teacher's answer.

In the reinforcement learning paradigm, however, there is no such a teacher who can tell the learning agent what it should have done. Instead, the learning agent will receive a reward or penalty for its action from the environment in which it is situated or from a critic who has some experience with the environment. Then, it adjusts its internal parameters to maximise the expected return in terms of its criteria of optimality. A learning process like this is *evaluative*. For evaluative learning, the learning agent can only know whether its action is good or bad from the reward or penalty but it can not know whether it is the right or the best action. For instance, a lion goes to place **A** and catches two zebras, but this does not mean that **A** is the best choice for the lion. The lion may have caught three zebras if it went to place **B**, or even more if it went to place **C**. On the other hand, it is also possible that the lion may have caught fewer zebras if

it went to place **B** or **C**. It stays uncertain until the lion has tried all places.

2.2.2 Exploration vs. exploitation

As a result of the nature of evaluative learning, reinforcement learning has to balance exploration and exploitation. In the reinforcement learning paradigm, *exploration* is to try actions the reward for which is unknown; on the other hand, *exploitation* is to choose the action which can offer the most rewards among all actions having been tried.

Although exploitation can utilise what has been discovered during exploration, it may miss the chance to find better solutions. On the other hand, though exploration offers the opportunity to find better solutions, it may waste the time of the agent on exploration and may even lead the agent to a worse situation. Consider the lion example that we have just discussed. If the lion always goes to place **A** (assume the environment does not change, viz. deterministic and stationary; of course, this is not the actual case because zebras can also learn the behavior of the lion), it can get two zebras every day and will miss the opportunity to get three zebras or even more daily in other places. But if it tries other places every day, it may go to some places where there is only one or even zero zebras and at the same time miss the opportunity to catch two zebras in place **A**. This is the dilemma of exploration and exploitation that has to be balanced in reinforcement learning.

2.2.3 Immediate vs. delayed rewards

In the reinforcement learning paradigm, rewards can be received immediately after the learning agent's action, a period after the learning agent's action (time delayed), or after a sequence of the learning agent's actions (action delayed). For instance, the n-armed bandit problem is a problem with immediate rewards and the chess problem is a problem with action sequence delayed rewards. For the n-armed bandit problem, if a reward is given only a period after an arm is pushed, the problem becomes one with time delayed rewards.

Delayed rewards, whether action delayed or time delayed, dramatically increase the difficulty for reinforcement learning. In particular, action delayed rewards give rise to the problem of temporal credit assignment, which will be discussed in section 2.2.4. For time delayed rewards, we need to balance immediate

and delayed rewards. Although delayed rewards are worth considering, they are not so important as immediate rewards in many tasks. In the lion case mentioned earlier, for example, it is more relevant for it to get food immediately than to wait for the food that will appear several days later. A good way to balance them is to use discounted rewards as discussed in the infinite-horizon discounted model in section 2.1.9. In addition, for agents who have limited time or limited energy budget, their optimal behavior not only depends on the expected return but also depends on their time limit or energy budget, which will be discussed in section 5.6.4.

2.2.4 Credit assignment

Another factor that makes reinforcement learning harder than supervised learning is that it is not easy to decide which actions among a sequence of actions contribute to the rewards received after the sequence of actions and what their contribution is. This is the problem of *credit assignment*, viz. how to distribute credit for success or rewards among many decisions all of which may have contributed to the success.

There are mainly two kinds of credit assignment problems, the structural credit assignment problem and the temporal credit assignment problem. In the reinforcement learning paradigm, the temporal credit assignment problem is more relevant than the structural credit assignment problem. Whereas the *structural credit assignment problem* occurs when there are several components of an action or an environment state that may give rise to a reward, the *temporal credit assignment problem* happens when there are a sequence of state-action pairs that lead to a reward, which is the usual case in reinforcement learning with delayed rewards. In a chess game, for instance, a reward is only given at the end of the game. Every state-action pair before the end of the game may or may not contribute to the result of the game (win or lose).

The temporal credit assignment problem has been successfully solved by temporal-difference (TD) learning [38], which will be discussed in section 2.3.4.

2.2.5 Designing a reward function

The problem of *designing a reward function*, or the *credit structuring problem* [39–41], is how to place rewards in appropriate states to attain the intended aim of

learning, viz. for the agent to behave as we expect. In reinforcement learning, the learning agent learns to maximise its return. So, the reward assignment effectively decides the direction or goal of learning. In other words, if one wants a learning agent to learn to accomplish a goal, one has to assign rewards properly to achieve the goal: when does it deserve a reward? How much should the reward be? For instance, if one wants a robot to press button **A** rather than button **B**, one should give it a (+1) reward if it presses button **A** and give it a (-1) penalty if it presses button **B**. On the other hand, if one wants the robot to press button **B**, one should give it a (+1) reward if it presses button **B** and give it a (-1) penalty if it presses button **A**.

Improper reward assignment may lead to slow learning or even undesired behaviours. For example, in a chess game, if one gives the learning agent a reward both when it takes a piece of its opponent and when it wins, it may find a way to maximise its return by taking the opponent's pieces even at the price of losing the game.

In the real world, most problems are too complex to define rewards properly, especially in a time- and space-continuous environment because there are too many situations to consider. Consider a recycling robot, for example, it should not only get a positive reward when it successfully collects a piece of rubbish, it should also get a negative reward when its battery is run out before reaching recharging sites, get a negative reward when it hits something and get a negative reward when it is hit by something. Besides, thousands of other either positive or negative reward assignments are needed to ensure that it behaves properly. In addition, the values of these rewards have to be carefully chosen. For example, if the recycling robot is given +1 every time it collects a piece of rubbish while given -10 every time it hits something, its priority will try to avoid hitting something rather than collecting rubbish. On the other hand, if the recycling robot is given +10 every time it collects a piece of rubbish while given -1 every time it hits something, it will try hard to collect rubbish even at the risk of hitting something.

2.2.6 Rewards vs. value

A *reward function* is defined as what an agent can receive immediately by taking a certain action in a particular state of the environment in which it is situated. Although a learning agent can get a high reward in one time step by following a

greedy policy based on the reward function, this cannot guarantee a high total of rewards in the future time steps, viz. a high return. That is, the greedy policy choosing the action which is optimal based on the one-step reward function does not guarantee optimality in the longer term. Suppose, if a learning agent takes action **A** in a certain state it will win a high reward (10 points). But this action will lead it to state **I** and none of all available actions in state **I** can give the agent a reward more than 3 points. In contrast, if the learning agent takes action **B** in the initial state, it will get a lower reward (5 points) and enter state **II**. But in the state **II**, it can receive a reward more than 10 points by taking any of available actions. In this case, the lower immediate reward will lead to a higher return.

In contrast, a *value function* is defined as the expected return that an agent can receive by following a policy from a particular state of the environment in which it is situated. Whereas a reward function indicates what is good in an immediate sense, a value function specifies what is good in the long run. If the value of an action in a certain state is the highest, taking the action in that state can ensure the highest return in the long run as long as the value function has been calculated correctly because the computation of the value function has already considered all possible future states, actions and rewards.

2.2.7 Generalisation

If a problem has only a small number of states and actions, it is possible to just use tables to record the estimated value for each state or for each state-action pair. If the problem is very large, however, it needs a lot of time and space to store data in a table and then read data from the table. Furthermore, many states or state-action pairs in the problem may have not been experienced before. In this case, no estimated values are available for these states or state-action pairs from the table. One way to solve these problems is to use generalisation that can not only avoid storing a great quantity of data but also predict the values of the states or state-action pairs which have not been experienced previously.

Generalisation is a typical supervised learning problem (structural credit assignment). A variety of techniques for supervised learning can be used to generalise the mapping from states to values or the mapping from state-action pairs to values. These techniques include neural networks [42], fuzzy logic [43] and local memory-based methods [44].

2.3 Methods for solving reinforcement learning problems

Because reinforcement learning problems are very common in the real world, many methods were proposed to solve them from research in optimal control, psychology, neuroscience and machine learning. Among them, there are mainly three fundamental classes of methods, viz. dynamic programming, Monte Carlo methods, and temporal-difference learning.

2.3.1 Bellman equations

The value function, whether the function of states or the function of state-action pairs, has recursive relationships, viz. the value ($V(s)$ or $Q(s,a)$) of one state (s) or state-action pair (s,a) can be expressed as a function of the value of possible successor states or state-action pairs. For the infinite-horizon discounted model, according to Sutton and Barto [2] for example, we can get

$$\begin{aligned} V^{\pi}(s) &= E_{\pi}(R_i|s_i = s) \\ &= E_{\pi}(\sum_{k=i+1}^{\infty} \gamma^{k-i-1} r_k | s_i = s) \\ &= E_{\pi}(r_{i+1} + \gamma \sum_{k=i+1}^{\infty} \gamma^{k-i-1} r_{k+1} | s_i = s) \\ &= \sum_{a \in \mathcal{A}(s)} \pi(s,a) \sum_{s' \in \mathcal{S}} \mathcal{P}^a_{ss'} [\mathcal{R}^a_{ss'} + \gamma E_{\pi}(\sum_{k=i+1}^{\infty} \gamma^{k-i-1} r_{k+1} | s_{i+1} = s')] \\ &= \sum_{a \in \mathcal{A}(s)} \pi(s,a) \sum_{s' \in \mathcal{S}} \mathcal{P}^a_{ss'} [\mathcal{R}^a_{ss'} + \gamma V^{\pi}(s')] \end{aligned} \tag{2.14}$$

where π is the policy of the agent, $\mathcal{P}$ is the probability distribution of the state transition, $\mathcal{P}^a_{ss'}$ is the probability that the next state is s' if the agent takes action a in state s. $\mathcal{R}^a_{ss'}$ is the expected value of the next reward that the agent will receive if it takes action a in state s and then enters the state s'. Equation 2.14 is called the *Bellman equation* for V^{π}. Similarly, we can get the *Bellman equation* for Q^{π} as follows [2].

$$Q^{\pi}(s,a) = \sum_{s' \in \mathcal{S}} \mathcal{P}^a_{ss'} [\mathcal{R}^a_{ss'} + \sum_{a' \in \mathcal{A}(s')} \pi(s',a') \gamma Q^{\pi}(s',a')] \tag{2.15}$$

There is always at least one policy, called an *optimal policy* denoted as π^*, which is better than or equal to all other policies. All optimal policies share the same value function, called the *optimal value function* denoted as V^* and defined as

$$V^*(s) = \max_{\pi} V^{\pi}(s). \tag{2.16}$$

Similarly, the *optimal state-action value function* Q^* is defined as

$$Q^*(s,a) = \max_{\pi} Q^{\pi}(s,a). \tag{2.17}$$

With the *Bellman equation* for V^π and *Bellman's Principle of Optimality* [6], according to Sutton and Barto [2], equation 2.16 can be expended to

$$V^*(s) \quad = \quad \max_{a \in \mathcal{A}(s)} \sum_{s' \in \mathcal{S}} \mathcal{P}^a_{ss'} [\mathcal{R}^a_{ss'} + \gamma V^*(s')]. \tag{2.18}$$

Similarly, with the *Bellman equation* for Q^π and *Bellman's Principle of Optimality*, according to Sutton and Barto [2], equation 2.17 can be expended to

$$Q^*(s,a) \quad = \quad \sum_{s' \in \mathcal{S}} \mathcal{P}^a_{ss'} [\mathcal{R}^a_{ss'} + \gamma \max_{a' \in \mathcal{A}(s')} Q^*(s',a')]. \tag{2.19}$$

Both equation 2.18 and equation 2.19 are called the *Bellman optimality equation.*

For each state or each state-action pair, there is one Bellman optimality equation (V^* or Q^*). There are also an equal number of unknowns. Therefore, if $\mathcal{P}^a_{ss'}$ and $\mathcal{R}^a_{ss'}$ are known, we can get V^* or Q^* by solving their Bellman optimality equations in principle. With V^* or Q^*, we can further easily determine an optimal policy by following the action a which offers the maximum expected return ($\operatorname{argmax}_{a \in \mathcal{A}(s)} \sum_{s' \in \mathcal{S}} \mathcal{P}^a_{ss'} [\mathcal{R}^a_{ss'} + \gamma V^*(s')]$ or $\operatorname{argmax}_{a \in \mathcal{A}(s)} Q^*(s,a)$) in any state s, viz. the greedy policy in terms of the expected return.

In practice, however, this method has limited use. Firstly, it assumes a finite Markov decision process. In real-world problems, the Markov property is rarely strictly satisfied. Many problems, however, can be approximated by MDPs when their states are carefully constructed and represented. Secondly, it requires the knowledge of the dynamics of the environment, viz. the probability distribution of the state transition $\mathcal{P}^a_{ss'}$ and the expected value of the next reward $\mathcal{R}^a_{ss'}$. It is usually the case that we do not know either of them before hand. Even if the previous two conditions are satisfied, the time and space required to solve the

equations when there are a lot of states may be prohibitive. Take the game of backgammon as an example: the first two assumptions are valid. But the game has about 10^{20} possible states and therefore it would take the fastest computers at present thousands of years and more than 100 Exabytes (10^{11} Gigabytes) memory to solve the equations.

2.3.2 Dynamic programming

In general, *dynamic programming* [6] refers to a technique that breaks a complex problem down to simpler subproblems and then builds the solution to the complex problem on solutions to these subproblems. In this way, repeated subproblems only need to be solved once. In the optimisation and reinforcement learning paradigms, it usually refers to methods that calculate the estimated value of a state based on the estimated values of all possible successor states and the probability of their occurrence. Here, two of these methods are discussed, viz. policy iteration and value iteration.

Policy iteration has two main stages, the policy evaluation stage and the policy improvement stage. At the policy evaluation stage, the method updates the estimated values of all states under a certain policy with the estimated value of all possible successor states using the Bellman equations iteratively. The updating process repeats until the difference in the estimated value of every state, before and after the estimated value of the state is updated, is less than a small positive number, viz. the estimated value function for the policy has converged/ become stable. Then, it enters the policy improvement stage. In this stage, the old policy is replaced with the policy which maximises the expected return in every state, viz. the greedy policy in terms of the value function. Next, the new policy is evaluated. The two stages repeat until the policy is stable, viz. the policy does not change at the policy improvement stage.

One drawback of policy iteration, however, is that it may take many iterations to evaluate one policy. The *value iteration* method addresses this problem by conducting only one sweep, viz. updating the estimated values of all states only once. The updated value function is then used to construct a new greedy policy.

Both methods converge to an optimal policy for finite MDPs in terms of the infinite-horizon discounted model [2, 45]. In addition, dynamic programming is much more efficient than solving Bellman optimality equations directly or searching the whole policy space (exhaustive search). Suppose that a problem has n

states and each state has m actions. The time complexity of dynamic programming for the problem is some polynomial function of m and n, whereas it takes the method of solving Bellman optimality equations directly or exhaustive search computational operations proportional to m^n. Despite its computational efficiency, however, dynamic programming still requires that the environment should be a finite MDP and the dynamics of the environment should be known.

2.3.3 Monte Carlo methods

Monte Carlo methods [3–5] refer to methods that use the actual or simulated samples to replace the actual probability distribution in computation. In the reinforcement learning paradigm, the dynamics of the environment, viz. the state transition and the value of the reward, is obtained from actual or simulated interaction with the environment by these methods. Therefore, these methods do not require the knowledge of the environment unlike dynamic programming.

There are on-policy and off-policy Monte Carlo algorithms. Here, however, we only introduce one on-policy Monte Carlo algorithm. Like dynamic programming, it also includes two stages, policy evaluation and policy improvement. An episode is first generated either through the actual interaction of the agent with the environment or through simulation using a non-greedy policy. The estimated value of every visited state-action pair during the episode is updated with the return following the occurrence of the state-action pair incrementally. If a state-action pair is visited more than once during the episode, we can either only consider the return following the first visit to the state-action pair (the *first-visit MC method*) or use the average of the returns following every visit to the state-action pair (the *every-visit MC method*). The new values of the state-action pairs are then used to improve the policy. Next, another episode is generated with the new policy and the process repeats until a termination condition is satisfied, e.g. when the policy is stable.

Unlike dynamic programming where all states and actions are considered, the state-action pairs experienced by the agent with Monte Carlo methods and a deterministic policy may not be complete and in fact quite sparse in many cases. In order to encourage exploration, a stochastic policy, e.g. ϵ-greedy, is normally used. In order for the learning to converge, however, the policy needs to move gradually towards a greedy policy. In the case of the ϵ-greedy policy, we need gradually reduce the value of ϵ to 0.

2.3.4 Temporal-difference (TD) learning

Temporal-difference (TD) learning [46] is an important technique for reinforcement learning that aims to eliminate the temporal differences in prediction. The purpose of TD learning is to solve prediction problems, viz. to learn a value function given a policy.

It is a combination of ideas from dynamic programming algorithms (bootstrap: estimate the value of the current state according to the values of successor states) and Monte Carlo methods (a model of the environment is not required: learn from experience).

Like dynamic programming algorithms, it also uses the estimated values of successor states to estimate the value of the current state. Unlike dynamic programming algorithms, however, it utilises experience (or samples) rather than a model of the environment to get the next state and the expected reward that it can earn when it transfers from the current state to the next state.

Like Monte Carlo methods, it also uses experience to find the next state and the consequential reward. Unlike Monte Carlo methods, however, it does not need to keep experiencing until the end of an episode before it starts learning because it uses the estimated value of the state one or more steps later and the rewards obtained during this period to update the estimated value of the current state.

Assume an agent is in the state s at present. Following a fixed policy π to take a certain action in s, the agent gets a reward r and then enters the next state s'. With TD learning [46], the value function of the state s can be updated with

$$V^{\pi}(s) \leftarrow V^{\pi}(s) + \alpha[r + \gamma V^{\pi}(s') - V^{\pi}(s)] \tag{2.20}$$

where $0 < \alpha \leq 1$ is the learning rate, $0 \leq \gamma \leq 1$ is the discount factor. If s' is a terminal state, $V^{\pi}(s') = 0$.

If α in the above equation satisfies

$$\sum_{n}^{\infty} \alpha(n) = \infty, \sum_{n}^{\infty} \alpha^2(n) < \infty \tag{2.21}$$

where $\alpha(n)$ is the value of α used in the n update of $V^{\pi}(s)$, the estimated $V^{\pi}(s)$ is guaranteed to converge to its true value [47] eventually. When $\alpha(n) = \frac{1}{n}$, it satisfies both conditions. When α is a fixed value, however, it cannot satisfy

both conditions at the same time. If it is equal to 0, the first condition is not satisfied. If it is bigger than 0, the second condition is not satisfied. But this may be a desirable behaviour to track nonstationary environments because the learning agent can forget past experience gradually by discounting them.

The learning rule 2.20 is based on the assumption that the expected value of the next state should be more accurate than that of the current state because the next state is closer to the end point which is usually where the most important reward lies and is also newer to reflect changes in the environment.

2.3.5 Q learning

One of the most important reinforcement learning algorithm is *Q learning* [7], an off-policy TD control algorithm. It updates the estimated value ($Q(s,a)$) of the current state-action pair (s,a) with the immediate reward received (r) and the maximum estimated value of all actions available ($\mathcal{A}(s')$) in the next state (s'), viz.

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a' \in \mathcal{A}(s')} Q(s',a') - Q(s,a)]. \tag{2.22}$$

If s' is a terminal state, $Q(s',*) = 0$ where $* \in \mathcal{A}(s')$. Similar to Monte Carlo methods, a stochastic policy is normally used for decision making.

Q learning is guaranteed to converge asymptotically given that each state-action pair is visited an infinite number of times and α decreases appropriately over time according to equation 2.21. Q learning is an *off-policy* learning algorithm in that the policy that it follows is different from the policy that it uses to evaluate values of state-action pairs. The policy that it follows is a non-greedy policy whereas the policy that it uses to evaluate values of state-action pairs is a greedy policy.

2.3.6 SARSA learning

In contrast to Q learning, SARSA learning [8] is an on-policy TD control algorithm. It updates the estimated value ($Q(s,a)$) of the current state-action pair (s,a) with the immediate reward received (r) and the estimated value ($Q(s',a')$) of the action (a') that the agent will take in the next state (s') under the current policy, viz.

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma Q(s',a') - Q(s,a)]. \tag{2.23}$$

Like Q learning, if s' is a terminal state, $Q(s', *) = 0$ where $* \in \mathcal{A}(s')$. Similar to both Monte Carlo methods and Q learning, a stochastic policy, is normally used for decision making. Unlike Q learning, however, SARSA learning is an *on-policy* learning algorithm in that the policy that it follows is the same with the policy that it uses to evaluate the values of state-action pairs.

2.3.7 Reinforcement comparison and actor-critic methods

The idea of *reinforcement comparison* [2,38] is to compare a reward for one action with a reference reward: if the reward (r) is higher than the reference reward, the action is more likely to be taken; otherwise, it is less likely to be taken. A straightforward choice of the reference reward is an average of all rewards previously received after the action is taken ($\bar{r}$).

Unlike most reinforcement learning algorithms which calculate the estimated values of actions, reinforcement comparison methods calculate a measure of the preference for each action instead. The preference for action a ($p(a)$) can be calculated by

$$p(a) \leftarrow p(a) + \beta[r - \bar{r}] \tag{2.24}$$

where

$$\bar{r} \leftarrow r + \alpha[r - \bar{r}] \tag{2.25}$$

and both α and β are small numbers.

With the preference function for actions, actions can be chosen according to a stochastic policy similar to other reinforcement learning algorithms.

It is worth noting, however, reinforcement comparison methods do not work with a sequence of actions. In these cases, a time difference error or a critic is needed to guide the agent instead of the difference between the current reward and the average reward. This kind of method is called the *actor-critic method*, addressing learning from delayed rewards in the absence of mechanisms for secondary reinforcement. This method updates the preference for action a ($p(s, a)$) in s with

$$p(s, a) \leftarrow p(s, a) + \beta\delta \tag{2.26}$$

where $\delta = r + \gamma V(s') - V(s)$, which is the same with TD learning.

2.3.8 Eligibility traces

The learning rule introduced in Section 2.3.4 is called TD(0). It only looks ahead one step to update state value estimates, which may cause a low speed of convergence. To overcome this drawback, the values of all states recently visited are updated according to their eligibility to learn when a reward is received. This kind of algorithm is called TD(λ). The eligibility of states to learn is calculated using an *eligibility trace* which is defined as:

$$e(s) \leftarrow \begin{cases} \gamma\lambda e(s) + 1 & \text{if } s \text{ is the current state} \\ \gamma\lambda e(s) & \text{otherwise} \end{cases} \tag{2.27}$$

where $0 \leq \lambda \leq 1$ and initially $e(s) = 0$ for all $s \in S$. When $\lambda = 0$, the rule becomes TD(0); when $\lambda = 1$ and $\gamma = 1$, it behaves exactly the same with a Monte Carlo method for an undiscounted episodic task.

The eligibility trace of one state is accumulated each time the state is visited and then fades away gradually when the state is not visited. This kind of eligibility trace is called an *accumulating trace*. One drawback of this kind of eligibility trace is that it is unbounded and may become greater than 1 when a state is visited again before its eligibility trace decays to 0. One way to overcome the drawback is to increase the eligibility trace of a state to 1 each time the state is visited and then fades away gradually when the state is not visited. This kind of eligibility trace is called a *replacing trace*.

With the eligibility trace, the learning rule of TD(λ) can be expressed as:

$$V(u) \leftarrow V(u) + \alpha[r + \gamma V(s') - V(s)]e(u) \tag{2.28}$$

where u is any of states visited between the last reward and this reward and $V(u)$ is the value of the state u, r is the value of the reward received in state s, and γ is the discount factor as before. That is, $V(u)$ is updated for all u.

In practice, TD(λ) learning often converges considerably faster for large λ than TD(0); but it is also more complex and needs more computational resources. In addition, it is worth noting that the idea of eligibility traces can also be used in other reinforcement learning algorithms, e.g. Q learning and SARSA learning.

2.3.9 Discussions of different methods

In order to converge with probability one to an optimal policy and the optimal value function of states or state-action pairs, these methods including adaptive/iterative dynamic programming, Monte Carlo and TD like algorithms need to update the estimated values of all states or state-action pairs an infinite number of times [2, 48–51]. Specifically for Monte Carlo and TD like algorithms, their policy needs to converge in the limit to the greedy policy due to their sampling feature and their learning rate needs to be gradually reduced according to equation 2.21 due to their incremental updating rule.

According to whether or not they use a model of the environment, learning algorithms can be grouped into model-based methods and model-free methods. For instance, adaptive dynamic programming algorithms [6] are model-based methods and Monte Carlo methods [3–5] are model-free methods. Reinforcement learning algorithms can be either model-based (e.g. prioritised sweeping) or model-free (e.g. Q learning and SARSA learning). Model-based reinforcement learning algorithms take advantage of the model learned during their interaction with the environment and can perform extra learning with the model. This is especially useful when the agent has not enough interaction with the environment or interaction with the environment is very costly. On the other hand, however, model-based reinforcement learning algorithms need more computational resources to learn a model of the environment and to perform the extra learning. In addition, they are not suitable for nonstationary environments since the learned model which may be a good representation of the old environment usually does not represent the new environment well.

According to whether or not the estimated value of one state or state-action pair is updated based on the estimated value of another state or state-action pair, learning algorithms can be divided into two groups, viz. bootstrap algorithms and non-bootstrap algorithms. Adaptive dynamic programming algorithms and TD(0) are bootstrap algorithms while Monte Carlo methods and TD(1) are non-bootstrap algorithms. Empirical data [2] show that bootstrap algorithms usually perform better than non-bootstrap algorithms. On the other hand, however, it is harder to use generation methods for bootstrap algorithms than non-bootstrap algorithms. In some cases, bootstrap algorithms cannot find optimal solutions. For instance, if a linear gradient-descent function approximator is used, on-policy bootstrap algorithms can only find solutions with a near minimal mean squared

error whereas non-bootstrap algorithms can find minimal mean squared error solutions [2]. In addition, Baird [52] further showed an example where off-policy dynamic programming methods become unstable for some initial values of the parameters when a gradient-descent function approximator is used.

According to whether or not the policy that the agent follows is the same with the policy that it uses to evaluate values of states or state-action pairs, learning algorithms can be grouped into on-policy learning and off-policy learning. Dynamic programming methods, Monte Carlo methods and reinforcement learning algorithms have both on-policy learning algorithms and off-policy learning algorithms. Empirical data [2] show that off-policy learning algorithms (e.g. Q learning) usually learn faster than on-policy learning algorithms (e.g. SARSA learning). On the other hand, however, the performance of off-policy learning algorithms may be worse than on-policy learning algorithms in some situations. For example, Sutton and Barto [2] showed a cliff-walking task where the optimal policy of Q learning walks along the cliff which results that the agent occasionally falls off the cliff and gets a big penalty whereas the optimal policy of SARSA learning walks along a safer path. It is worth noting, however, that both methods would asymptotically converge to the same optimal policy if they change their policy gradually from a non-greedy policy to a greedy policy. In a dynamic environment, however, a non-greedy policy is needed all the time.

Chapter 3

Related work

In the last chapter, the fundamental knowledge of reinforcement learning is provided. In this chapter, we will further discuss recent research on reinforcement learning which are related with our work.

This chapter is organised as follows. In section 3.1, the biological motivation for reinforcement learning, classical conditioning and its simulation by TD learning are discussed. In section 3.2, we introduce a simple and popular reinforcement learning problem, viz. the n-armed bandit problem, and then discuss some of its classical solutions. Approaches to estimation of variance are present in section 3.3. Section 3.4 reviews recent research on reinforcement learning in semi-Markov decision processes. Section 3.5 reviews recent research on reinforcement learning in dynamic environments. In the last section, the relationship of our research to existing research on reinforcement learning in dynamic environments is discussed.

3.1 Classical conditioning

Classical conditioning [53] is a simple form of reinforcement learning and also a motivation for temporal-difference (TD) learning, an important reinforcement learning algorithm. It involves various behavioural phenomena that animals learn and predict rewards in their environment (e.g. Pavlovian classical conditioning) and also respond to changes in the environment (e.g. the disappearance of a reward — the extinction phenomenon in classical conditioning). Its study begins with Pavlov's experiments on dogs in the 1890s. Normally, dogs will salivate (an unconditional response) when they receive food (an unconditional stimulus).

After a dog is repeatedly fed just after a bell (a conditional stimulus) is rung, it salivates (a conditional response) whenever the bell sounds (Pavlovian classical conditioning). In the reinforcement learning terms, the dog is a learning agent, food is a reward, the ringing of the bell is a state of the environment, and the conditioning is a prediction of the reward. Following Pavlov's experiments, a great number of scientists did similar experiments and found a variety of other phenomena [54–56].

Since the discovery of classical conditioning, many researchers have sought to explain the phenomena. In 1972, Rescorla and Wagner [36] proposed the Rescorla-Wagner rule which has successfully explained Pavlovian, extinction and blocking phenomena of classical conditioning, but failed to explain secondary conditioning. In 1988, Sutton and Barto [37] used temporal-difference (TD) learning to provide a successful explanation of various phenomena of classical conditioning including secondary conditioning, which will be discussed in detail in section 3.1.2. Malaka [57] reviewed various models developed for simulating classical conditioning and classified them into two groups, viz. trial-based models (e.g. the Rescorla-Wagner model [36]), where weights are only updated at the end of one trial, and real-time models (e.g. the Sutton-Barto model [37]), where weights are updated in real time during one trial. However, all these models used high-level abstractions of neurons which ignore the temporal dynamics of real neurons. Classical conditioning, however, is dynamic in essence because it mainly learns and predicts the dynamic environment. The real-time models solve this problem by using a temporal learning rule instead of directly utilising the temporal dynamics of biological neurons.

3.1.1 Phenomena of classical conditioning

There are various phenomena of classical conditioning. But here we only discuss Pavlovian, Extinction, Partial, Blocking, Inhibition, Overshadowing and Secondary phenomena of classical conditioning.

1. *Pavlovian*

   If an animal is given a reward repeatedly shortly following a conditional stimulus, the animal will be able to anticipate the reward when the conditional stimulus occurs. This phenomenon is called Pavlovian classical conditioning.

2. *Extinction*
   After the conditioning of the conditional stimulus with the Pavlovian classical conditioning procedure, if the reward is removed, the association between the conditional stimulus and the reward will be gradually lost. This phenomenon is called extinction in classical conditioning.

3. *Partial*
   If two training experiments, in one of which a reward follows a conditional stimulus and in the other of which no reward follows the same conditional stimulus, are alternated repeatedly, the animal will anticipate a partial or weakened expectation of the reward. This phenomenon is called partial conditioning.

4. *Blocking*
   After the conditioning of a conditional stimulus, if another stimulus is presented at the same time with the first conditional stimulus, the association between the first conditional stimulus and the reward will not change and will block the formation of the association between the second conditional stimulus and the reward. This phenomenon is called blocking in classical conditioning.

5. *Inhibition*
   If two experiments, in one of which a reward follows when only one conditional stimulus (the first stimulus) appears and in the other of which no reward follows when both the stimulus (the first stimulus) and another conditional stimulus (the second stimulus) appear together, are alternated repeatedly, an expectation of reward will be associated with the first conditional stimulus and the suppression of an expectation of reward will be associated with the second conditional stimulus. This phenomenon is called inhibition in classical conditioning.

6. *Overshadowing*
   If two conditional stimuli are presented at the same time shortly preceding a reward, each of them will be partly associated with an expectation of reward. This phenomenon is called overshadowing in classical conditioning.

7. *Secondary*

After the conditioning of the first conditional stimulus, if the reward is removed and another stimulus is presented before the first conditional stimulus, the first conditional stimulus will gradually lose its association with an expectation of reward and the second conditional stimulus will gradually form its association with an expectation of reward but will still lose its association in the end. This phenomenon is called secondary conditioning.

In addition to classical conditioning, there is another kind of conditioning, viz. instrumental conditioning. Unlike the conditioning discussed above, in the process of instrumental conditioning, the actions of animals can have an influence on how many rewards they can obtain, which is more usual in the real world.

3.1.2 Simulation of classical conditioning

The phenomena of Pavlovian conditioning, partial conditioning, inhibitory conditioning, overshadowing and extinction demonstrate that animals can associate stimuli with rewards and penalties. Moreover, other phenomena including secondary conditioning further demonstrate that animals can also learn the relative occurrence order of stimuli and rewards. The Rescorla-Wagner rule [36] cannot realise secondary conditioning because it completely ignores the timing information of stimuli and rewards. In order to overcome this difficulty, Sutton and Barto [37] extended the Rescorla-Wagner rule to temporal-difference learning with the aim to eliminate the time difference between the actual and predicted total future rewards rather than only to eliminate the amount difference between the actual and predicted total future rewards.

Suppose there are n time-dependent stimuli, $s_1(t)$, $s_2(t)$,...,$s_n(t)$. For any of the stimuli $s_i(t)$, its eligibility trace (eligibility to learn) $e_i(t+1)$ at time $t+1$ can be expressed as:

$$e_i(t+1) = e_i(t) + \lambda\,[u_i(t) - e_i(t)] \tag{3.1}$$

where $e_i(0) = 0$, λ is a positive constant, and

$$u_i(t) = \begin{cases} 1 & \text{if } s_i \text{ is present at time } t \\ 0 & \text{otherwise.} \end{cases}$$

The predicted total future rewards $v(t+1)$ at the time $t+1$ can be expressed

as:

$$v(t+1) = \left\lfloor \sum_i w_i(t) e_i(t+1) \right\rfloor \tag{3.2}$$

where

$$\lfloor x \rfloor = \begin{cases} x & \text{if } x \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

and, $w_i(t)$ is the weight of the stimulus $s_i(t)$ at time t and $i = 1, 2, \ldots, n$.

The TD rule is to update $w_i(t+1)$ from $w_i(t)$ with the aim to minimise the time difference between the actual and predicted total future rewards. It can be expressed as:

$$w_i(t+1) = w_i(t) + \beta \delta(t+1) \alpha_i e_i(t+1) \tag{3.3}$$

where α_i is a positive constant depending on the conditional stimulus $s_i(t)$, β is a positive constant depending on the unconditional stimulus, $\delta(t+1)$ is the approximate difference between the the actual and predicted total future rewards at time t and can be expressed as:

$$\delta(t+1) = r(t+1) + \gamma v(t+1) - v(t) \tag{3.4}$$

where $r(t+1)$ is the reward received at the time $t+1$,

$$r(t+1) = \begin{cases} 1 & \text{if } r \text{ is present at time } t+1 \\ 0 & \text{otherwise} \end{cases}$$

and γ is the discount rate as before. Substituting v with equation 3.2, we can get

$$\delta(t+1) = r(t+1) + \gamma \left\lfloor \sum_i w_i(t) e_i(t+1) \right\rfloor - \left\lfloor \sum_i w_i(t) e_i(t) \right\rfloor . \tag{3.5}$$

3.1.3 Neural substrate of TD learning

Although temporal-difference learning has successfully simulated various phenomena of classical conditioning, it is still unknown how animals learn to predict future rewards. If animals also use temporal-difference learning, how do they get the time difference information between the actual and predicted total future rewards, which is a critical factor to temporal-difference learning?

A series of studies by Schultz et al. [58, 59] showed that the activity of the

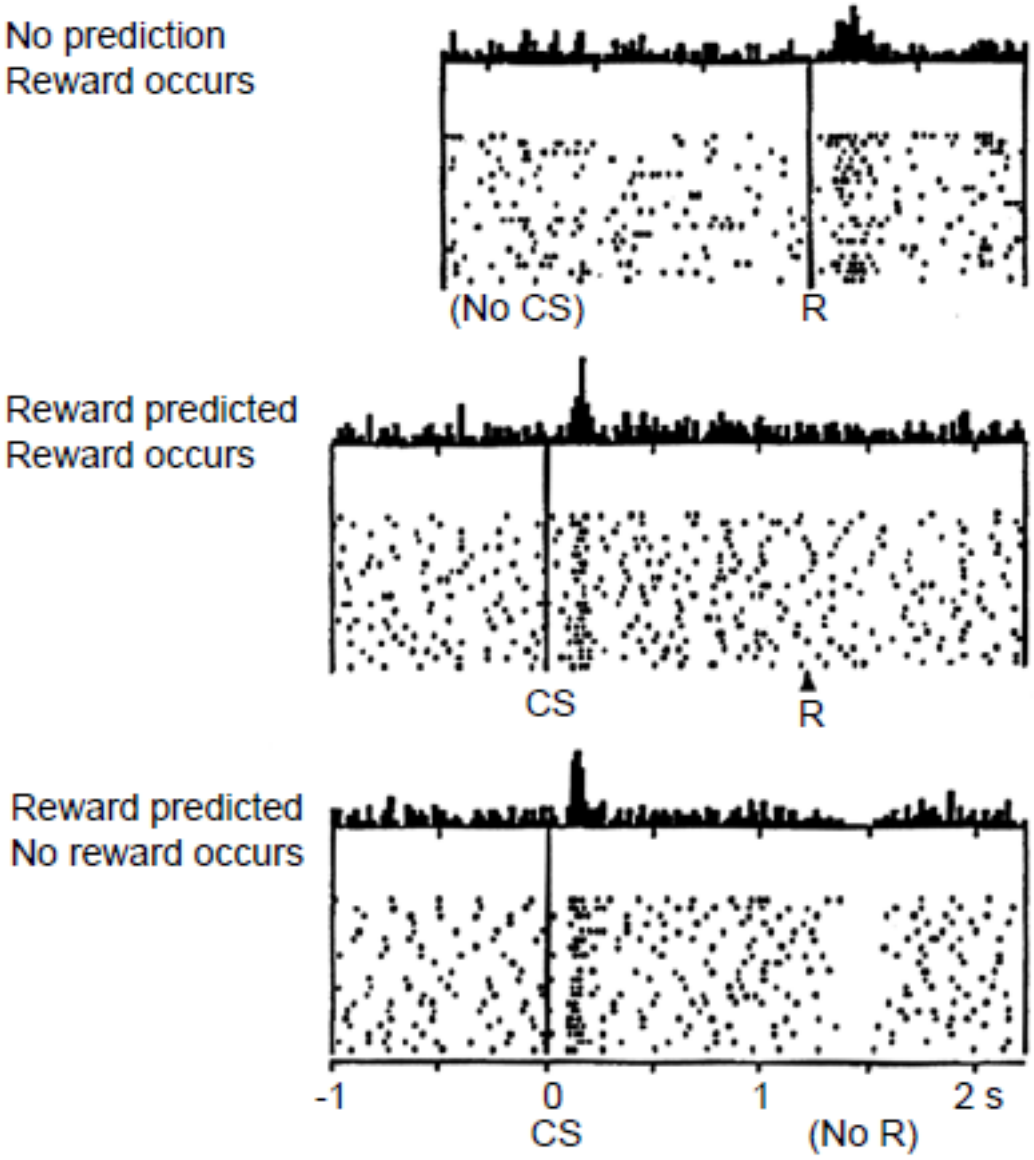

Figure 3.1: The activity of the dopaminergic neurons during classical conditioning experiments (taken from [58]). Before learning, when a reward occurs unexpectedly, the dopaminergic neurons are activated at the time just after the reward; after learning, when a conditional stimulus predicts the reward, the activation time of the dopaminergic neurons is moved forwards to the time just after the conditional stimulus; after learning, if the reward fails to happen, the activity of the dopaminergic neurons will be depressed at the time just after the reward was expected.

dopaminergic neurons in the ventral tegmental area (VTA) in the midbrain performs the same function as the prediction error δ in temporal-difference learning as Figure 3.1 shows. Before learning, when a reward occurs unexpectedly, the dopaminergic neurons are activated at the time just after the reward; after learning, when a conditional stimulus predicts the reward, the activation time of the dopaminergic neurons is moved forward to the time just after the conditional stimulus; after learning, if the reward fails to happen, however, the activity of the dopaminergic neurons will be depressed at the time just after the reward is expected.

Furthermore, if signals produced from dopaminergic neurons can be used to modify the plasticity of the neurons involved in reinforcement learning, temporal difference learning will be able to be realised in a biological neural system.

3.2 The n-armed bandit problem

The *n-armed bandit problem* is probably the simplest and most studied reinforcement learning problem. It has only one state but n actions. Many mathematicians and engineers have studied the problem extensively [60–63] and invented many novel methods which have formed the foundations of reinforcement learning and optimal control.

3.2.1 The problem

The n-armed bandit problem can be described as follows. There is an n-armed bandit and the agent can push any of its arms. The probability distributions of the payoffs of arms are unknown. After one arm is pushed, a reward (r) is returned. After receiving a reward, the agent goes back and starts again.

When the possible value of r is either 1 representing success/win or 0 representing failure/loss, the problem is a *binary n-armed bandit problem.* On the other hand, if the possible value of r is any real value, the problem is a *real-valued n-armed bandit problem.*

The n-armed bandit problem can be either treated as a one-step episodic task, a multiple-step episodic task, or a continual task. If it is treated as a one-step episodic task, each episode starts when one arm is pushed and ends when a reward (r) is received. The goal is the same in terms of all three criteria of optimality

discussed previously, viz. to maximise the expected one-step return

$$E(R) = E(r). \tag{3.6}$$

If it is treated as a multiple-step episodic task or a continual task, on the other hand, the goal is quite different for the three criteria of optimality.

Although this problem has only one state, in a multiple-step episodic task or a continual task, the policy of the agent also depends on the number of remaining plays and previous results for multiple-step episodic tasks and continual tasks. Therefore, in order for the problem to be treated as a MDP, the state needs to be augmented with these signals $(i_1, j_1, \ldots, i_n, j_n)$ where i_k is the number of times arm k has been chosen and j_k is the total value of rewards received from arm k. The augmented states are called *belief states*.

3.2.2 Dynamic programming

Although dynamic programming can be used to solve both binary n-armed bandit problems and real-valued n-armed bandit problems, we only consider a binary n-armed bandit problem here for simplicity.

Because dynamic programming requires the knowledge of the probability distributions of the payoffs of arms, which are unknown, we assume a prior distribution for each arm initially, e.g. a uniform distribution between 0 and 1. The estimation of the probability distribution for each arm can then be updated with experience using Bayes' rule.

For simplicity, only the finite-horizon model is considered here. Denote the length of the horizon of the finite-horizon model by o, viz. the total number of times that the agent can play. Denote the value of the optimal policy by V^*. Then

$$V^*_m(i_1, j_1, \ldots, i_n, j_n) = \max_k \left[\begin{array}{l} p_k V^*_{m-1}(i_1, j_1, \ldots, i_k + 1, j_k + 1, \ldots, i_n, j_n) + \\ (1 - p_k) V^*_{m-1}(i_1, j_1, \ldots, i_k + 1, j_k, \ldots, i_n, j_n) \end{array} \right] \tag{3.7}$$

where p_k is the posterior subjective probability of arm k paying off given i_k, j_k and our prior probability, m ($m = o - \sum_k i_k$) is the number of times left to play. For the uniform priors, the posterior probability distribution is a beta distribution with the expected value $\hat{p_k} = \frac{j_k+1}{i_k+2}$ according to Laplace's rule of

succession [64–66]. The base case of the recursion is

$$V_1^*(i_1, j_1, \ldots, i_n, j_n) = \max_k p_k \tag{3.8}$$

There is a similar equation for each belief state. The number of equation is linear in the number of belief states times actions while the number of belief states is exponential in the horizon. In principle, dynamic programming can be used to find an optimal policy for any bandit problem. For large horizons, however, it is not practical. It is worth noting, however, if the criterion of optimality is the finite-horizon model, dynamic programming is the only technique that guarantees optimality.

3.2.3 Gittins allocation indices

For the infinite-horizon discounted model, Gittins allocation indices can be used to find the optimal action that the agent should take in each step. *Gittins allocation indices* for some discount factor are tables of the number of times (i) that one arm has been chosen and the total value of rewards (j) received from the arm, viz. $I(i, j)$. The indices are obtained from dynamic programming methods. At each step, the agent just chooses the action which has the maximum index value, viz.

$$\operatorname*{argmax}_k I(i_k, j_k). \tag{3.9}$$

In this way, the optimality in terms of the infinite-horizon discounted model is guaranteed. It is worth noting that this method works not only with binary bandit problems but also with real-valued bandit problems. However, it only works under the infinite-horizon discounted model [1, 63] and has not yet been extended to more general reinforcement problems with delayed rewards.

3.2.4 Reinforcement learning

A typical reinforcement learning approach to solve the n-armed bandit problem is to update the estimated value (Q) of one action (a) with the reward (r) received following the action and the previous estimated value of the action, viz.

$$Q(a) \leftarrow Q(a) + \alpha[r - Q(a)] \tag{3.10}$$

where α is the learning rate, the same as before.

A non-greedy policy, e.g. ϵ-greedy, is usually used for decision making. The optimal target of such algorithms is to maximise the expected return in one step, viz. the finite-horizon model, if we consider the n-armed bandit problem as an one-step episodic task, or to maximise the expected long term average return, viz. the average-reward model, if we consider the n-armed bandit problem as a continual task or a multiple-step episodic task.

3.3 Approaches to estimation of variance

The variance of a random variable X is defined as

$$Var(X) = E[(X - \mu)^2] \tag{3.11}$$

where μ is the expected value of X.

Extending the above equation with $\mu = E(X)$, we can get

$$Var(X) = E(X^2) - [E(X)]^2. \tag{3.12}$$

Usually, however, the probability distribution of X is unknown. If an entire population of X is known, denoted as $x_1, x_2, \ldots, x_n$, the variance of X can be calculated from the population variance δ^2, viz.

$$\delta^2 = \frac{\sum_{i=1}^{n} x_i^2}{n} - \frac{(\sum_{i=1}^{n} x_i)^2}{n^2}. \tag{3.13}$$

However, the population is usually not known, either. In this case, we have to estimate the population variance from a finite sample of observations denoted as $x_1, x_2, \ldots, x_k$, viz.

$$s^2 = \frac{\sum_{i=1}^{k} x_i^2}{k-1} - \frac{(\sum_{i=1}^{k} x_i)^2}{k(k-1)} \tag{3.14}$$

where s^2 is sample variance. Equation 3.14 is an unbiased estimator of the population variance because its expected value $E(s^2)$ is equal to the true variance of the population.

One drawback of equation 3.14 is that all samples have to be store. In addition, it is not suitable when the variance needs to be calculated online. For example, if the sample is available one by one and the variance needs to be estimated

whenever a new sample is available, which is the case for reinforcement learning, we have to repeat the calculation of variance and it is very costly. In order to solve these problems, we need an incremental on-line algorithm.

According to Knuth [67], the variance of a random variable X can be unbiasedly estimated from a finite sample of observations incrementally using

$$\begin{aligned} v_k &= \frac{(x_k - m_{k-1})(x_k - m_k) + v_{k-1}(k-2)}{k-1} \\ &= v_{k-1} + \frac{(x_k - m_{k-1})(x_k - m_k) - v_{k-1}}{k-1} \end{aligned} \tag{3.15}$$

where m is the estimated mean, v is the estimated variance, x_k is the k^{th} observed data, and m_k and v_k are the k^{th} estimation.

For bootstrap methods in an environment with multiple states, however, it is worth noting that unless a full dynamic programming backup is used, the estimated mean is non-stationary during the learning period. This will lead to a biased estimate of the variance in the value function [68], which in turn affects the performance of methods relying on unbiased statistics [69].

3.4 Reinforcement learning in semi-MDPs

Most classical reinforcement learning algorithms were developed for Markov decision processes. Semi-Markov decision processes pose a challenge to reinforcement learning. In this section, recent research in semi-MDPs in the reinforcement learning paradigm is reviewed.

3.4.1 Bellman equations for semi-MDPs

For the infinite-horizon discounted model, the target of the agent in SMDPs is to maximise

$$E(\int_0^\infty e^{-\beta t}\rho(s(t), a(t))dt) \tag{3.16}$$

where $x(t)$ and $a(t)$ are respectively the state and action at t, $\rho(s(t), a(t))$ is the reward rate after $a(t)$ is taken in $s(t)$ until the next state, $\beta > 0$ is a parameter, and $e^{-\beta t}$ serves as the discount factor.

The *Bellman equation* for a SMDP in terms of V^π under the infinite-horizon

discounted model is

$$V^{\pi}(s) = \sum_{a \in \mathcal{A}(s)} \pi(s,a) \sum_{s' \in \mathcal{S}} \mathcal{P}^{a}_{ss'} [\int_0^{\infty} \int_0^{t} e^{-\beta z} \rho(s,a) dz dF^{a}_{ss'}(t) + V^{\pi}(s') \int_0^{\infty} e^{-\beta t} dF^{a}_{ss'}(t)] \quad (3.17)$$

where π is the policy of the agent, $V^{\pi}(s)$ and $V^{\pi}(s')$ are respectively the values of s and s' under the policy π, $\mathcal{P}$ is the probability distribution of the state transition, $\mathcal{P}^{a}_{ss'}$ is the probability that the next state is s' if the agent takes action a in state s. $F^{a}_{ss'}(t)$ is the cumulative probability distribution function of the time taken by the state transition from s to s' when action a is taken in s.

Let $\mathcal{R}^{a}_{ss'} = \int_0^{\infty} \int_0^{t} e^{-\beta z} \rho(s,a) dz dF^{a}_{ss'}(t)$ and $\gamma^{a}_{ss'} = \int_0^{\infty} e^{-\beta t} dF^{a}_{ss'}(t)$. Equation 3.17 can be simplified to

$$V^{\pi}(s) = \sum_{a \in \mathcal{A}(s)} \pi(s,a) \sum_{s' \in \mathcal{S}} \mathcal{P}^{a}_{ss'} [\mathcal{R}^{a}_{ss'} + \gamma^{a}_{ss'} V^{\pi}(s')]. \quad (3.18)$$

Equation 3.18 is quite similar to its counterpart equation for MDPs (equation 2.14). The Bellman optimality equation for a SMDPs in terms of V^{π} under the infinite-horizon discounted model is

$$V^{*}(s) = \max_{a \in \mathcal{A}(s)} \sum_{s' \in \mathcal{S}} \mathcal{P}^{a}_{ss'} [\mathcal{R}^{a}_{ss'} + \gamma^{a}_{ss'} V^{\pi}(s')] \quad (3.19)$$

where V^{*} is the optimal value function.

3.4.2 Research in semi-MDPs

Bradtke and Duff [70] extended classical reinforcement learning algorithms developed for Markov decision processes, e.g. TD learning and Q learning, to semi-Markov decision processes under the infinite-horizon discounted model. Das and Gosavi [71] introduced a new model-free reinforcement learning algorithm, Semi-Markov Average Reward Technique (SMART), to solve SMDPs problems under the average-reward model.

In addition, Sutton et al. [72, 73] introduced the theory of options to bridge the gap between MDPs and semi-MDPs. In semi-MDPs, temporally extended actions or state transitions are considered as indivisible units, so there is no

way to examine or improve the structures inside the extended actions or state transitions. The options theory, however, considers temporally extended actions or state transitions, so called options, as temporal abstractions of an underlying MDP. In doing so, it offers the flexibility to represent problems at multiple levels of temporal abstractions, the potential to speed up planning and learning, and the possibility of modifying (constructing or decomposing) options and changing the course of temporally extended actions or state transitions by examining and changing the structures of options.

3.5 Reinforcement learning in dynamic environments

The environment may be stationary or nonstationary. In a stationary environment, the reward structure and transitions between states either do not change over time or vary with respect to a stationary probabilistic distribution. There are two kinds of stationary environments. The first one is a deterministic stationary environment. In a deterministic stationary environment, the reward structure and transitions between states (the state structure) do not change over time. Therefore, if a fixed and deterministic policy is used, one visit to every state-action pair is enough to learn their actual values under the policy. The other one is a stochastic stationary environment. In a stochastic stationary environment, the reward structure and transition between states change over time with respect to a probabilistic distribution and therefore the learning agent needs to visit every state-action pair many times before the true mean of their value is learned (estimating the expected value or the population mean from the sample mean) under the policy even if a fixed and deterministic policy is used. In both of these environments, however, the learning agent can just follow the optimal action in every state and does not need to continue its exploration or even continue learning once it has learned the true mean of the optimal value of every state-action pair because the environment will not change later. In a stationary environment, most classical value estimation reinforcement learning algorithms, e.g. TD learning [46], Q learning [48] and SARSA learning [74], converge with probability one to an optimal policy and the optimal value function as long as all state-action pairs are visited an infinite number of times, their policy converges in the limit to a greedy policy, and their learning rate is gradually reduced towards

0 as discussed previously.

In a nonstationary environment, the reward structure and transitions between states change over time and the change is not subject to a probabilistic distribution, viz. the probability that a state transition happens and the probability that a certain amount of reward is received for the state transition may change over time. The learning agent has to keep learning and exploring the environment because an optimal policy in the old environment may not be optimal in the new environment. Depending on the speed of changes in the environment, the nonstationary environment can be classified into two kinds of environments, viz. slowly changing environments and rapidly changing environments. In a slowly changing environment, the reward structure and transitions between states change slowly over time. In this kind of environment, most classical algorithms with a non-greedy policy and a fixed learning rate, e.g. Q learning and SARSA learning, can still handle the changes. However, there is little theoretical analysis about it [1]. In a rapidly changing environment, the reward structure and transitions between states change rapidly over time. In this kind of environment, a big discount factor might help to forget the past environment and therefore help to learn the current environment. However, it would become intractable if the environment keeps changing too rapidly because the environment has already become a different one even before the old one is learned and therefore the learning agent can never learn the environment.

In addition, there are two kinds of special nonstationary environments which are commonly encountered in real-world applications and that can also be better solved by utilising their special features, i.e. switching environments and cyclical environments. In a switching environment, the reward structure and transitions between states stay the same for a long time, then change abruptly, and then hold for a long time before another change. Real-world examples of this kind of environment include the closing of a highway road, the exhaustion of a water or food source, and a change or damage in the learning agent itself. In this kind of environment, though classical algorithms can also eventually learn the change through continuous exploration, the learning process is quite slow because it takes the learning agent a long time to unlearn the previous optimal policy [27]. Experiments conducted by Anderson et al. [28, 29] further show that in some cases it is even better to throw away the existing policy and start over than to continue with the existing policy. In a cyclical/recurrent environment, different

reward structures and transitions between states appear repeatedly. In this case, it may be better to store the mapping of the environment and learning parameters (e.g. the Q values for Q learning), and then recall the corresponding learning parameters when an stored environment reappears [75, 76].

Generally, a *dynamic environment* refers to a nonstationary environment, though some researchers also consider a stochastic stationary environment as a kind of dynamic environment. As discussed above, dynamic environments pose a big challenge to reinforcement learning algorithms. The following subsections will present a review of existing research on reinforcement learning in dynamic environments.

3.5.1 A fixed learning rate and finite time window

With a fixed learning rate α, the i^{th} past experience is discounted by $(1-\alpha)^{k-i}$ as shown in

$$Q_k = Q_{k-1} + \alpha(r_k - Q_{k-1}) = \alpha r_k + (1-\alpha)Q_{k-1} = (1-\alpha)^k Q_0 + \sum_{i=1}^{k} \alpha(1-\alpha)^{k-i} r_i \tag{3.20}$$

where k is the total number of experiences so far, Q_0 is the initial estimation of the value of the reward, Q_k is the k^{th} estimation of the value of the reward, r_i is the value of the i^{th} reward received, and α is the fixed learning rate. Therefore, the further away from the present, the more heavily the experience is discounted, which is called an *exponential recency-weighted average* [2].

It is worth noting that the value of α needs to be chosen carefully. With a large α, past experience is heavily discounted. This would make the learning prone to noise and may lead to instability because the learning is dominated by recent experience. On the other hand, with a small α, learning would become very slow and greatly depend on the initial value.

With a finite time window, past experience outside the time window is completely ignored and only experiences within the time window contribute to the learning. Similar to the choice of the value of α, the size of a finite time window should also be carefully picked. With too small a size, it would never learn anything. On the other hand, however, too large a size would lead to a slow response to changes in the environment.

These are general strategies for nonstationary environments and can be integrated into almost all reinforcement learning algorithms. However, they both discount past experience whether the environment changes or not. If the environment changes, they would help forget past experience gained in the old environment. If the environment does not change, however, they would limit the achievable learning because only part of the experience is used to learn and would make it impossible for the learned value to converge to its true value in a stationary but stochastic environment. Besides, the degree of discount (the value of α and the size of the finite time window) should be carefully chosen.

3.5.2 Non-greedy decision making

With non-greedy decision making, the learning agent deliberately chooses non-optimal actions with a certain probability so that it keeps exploring the environment and modifies its policy to suit the current environment.

Probably the simplest method to make non-greedy decisions is *ϵ-greedy* [7,77]. The optimal action is chosen most of the time but with probability ϵ actions are chosen randomly, regardless of their values.

Although ϵ-greedy is simple and effective, it chooses all actions with equal probability when it explores the environment (with probability ϵ). It may be more desirable to explore the actions with higher values than the actions with lower values. This idea leads to *softmax methods* [78, 79]. One of the most widely used softmax methods is *Boltzmann-Gibbs rule* which chooses action a with probability

$$\frac{e^{Q(a)/T}}{\sum_i e^{Q(i)/T}} \tag{3.21}$$

where T is a positive number. With a big T, the difference in the probability of being chosen between actions is very small. The net effect is close to uniformly random choice and therefore it encourages exploration. On the other hand, with a small T, the difference in the probability of being chosen between actions becomes big. The net effect is close to greedy choice and therefore it encourages exploitation.

In a stationary environment, T is set to a big value initially to encourage exploration and then a small value to reduce the cost of exploration. In a nonstationary environment, however, T should stay at a relatively big value to maintain exploration. In fact, from the point of view of the agent, the environment also

changes at the initial learning stage even though the environment is stationary and has not changed because its initial estimation of the environment is usually incorrect.

These non-greedy decision making methods are general strategies for nonstationary environments and are useful in the learning stage for stationary environments. They are widely used by almost all reinforcement learning algorithms. However, many algorithms use them whether the environment changes or not. It is preferable to use a big exploration rate when the environment has changed and a small one when the environment has not because exploration is costly. Furthermore, only when suboptimal actions have improved does increasing the exploration rate (e.g. ϵ for ϵ-greedy) help. In some cases, for example, when the optimal action has improved or suboptimal actions have worsened, increasing ϵ would make the situation worse. In addition, some states may satisfy the conditions of increasing ϵ whereas others do not. Therefore, a different ϵ for each state would be helpful to ensure that only the ϵ of the states that satisfy the conditions is increased when the environment changes.

3.5.3 Exploration bonuses

Sutton [27] introduced two kinds of problems in a navigation task which classical value estimation reinforcement learning algorithms struggle to solve.

The first problem is called blocking. It is worth noting that it is different from the blocking phenomenon in classical conditioning discussed previously. In this problem, after the learning agent finds the optimal path, the optimal path is blocked. Although classical value estimation algorithms can eventually learn about the blocking and find the new optimal path, they bump into the barrier again and again, and spend many trials to unlearn the previous optimal policy.

The second problem is called a shortcut. In this problem, after the learning agent finds the optimal path, a path, which is even shorter than the optimal one but which was previously closed, is opened. Classical value estimation algorithms may still follow the previous optimal path and never learn the shortcut because the previous policy/ value function keeps pushing the learning agent towards the previous optimal path even when it is away from the previous optimal path by a very small chance (e.g. ϵ-greedy).

In order to solve these two problems, Sutton suggested exploration bonuses [27] to encourage exploration of states which have not been visited for a long time.

In particular, it adds an exploration bonus to the reward that the learning agent receives by taking one action. In Sutton's paper [27], the amount of bonus is proportional to the square root of the number of time steps that have elapsed since the action was tried last time. Therefore, the longer the actions have not been tried, the bigger bonus reward they will get.

Although this method of exploration bonuses has dramatically improved the performance of classical value estimation reinforcement learning algorithms on the blocking and shortcut problems, it has several drawbacks. Firstly, the calculation of exploration bonuses is a bit heuristic and there is little theoretical analysis about it as the author of [27] has also pointed out. Secondly, the exploration bonus is undirected because all the actions with the same number of elapsed time steps will get the same amount of bonus. It may be preferable that the actions which may lead to a better path will get a bigger bonus reward. Thirdly, the bonus reward is potentially unbounded and may be very big and even overshadow the real reward. Finally, it will encourage exploration whether the environment changes or not.

Dayan and Sejnowski [80] further systematically extended the exploration bonus method by modelling the uncertainty of the way the environment changes and then turning it into exploration bonuses for uncertain states. The learning agent assumes that changes in the environment (for instance, the probability that a barrier is present in the maze problem) follow a probabilistic distribution. Compared with Sutton's exploration bonus method, it is more directed in the sense that it explores the states which potentially lead to better results if the situation changes (for instance, a barrier is opened in the maze problem) more.

Dayan and Sejnowski's extended exploration bonus method, however, would still encourage exploration whether the environment changes or not. Another problem is due to the fact that the learning is incremental. Suppose that the learning agent has found that a previous suboptimal action in one state has improved thanks to the exploration bonus. After one update, it is likely that its estimated value is still suboptimal. After that, however, it is less likely to be visited because it has lost the exploration bonus after being visited. In addition, it also needs a model of the environment and assumes that the environment changes with respect to a certain probabilistic distribution.

3.5.4 Interval estimation algorithm

The Interval Estimation (IE) algorithm [69] computes the confidence intervals of the estimated values of actions and then selects the action with the largest upper interval boundary instead of the one with the largest mean. In this way, it encourages the agent to choose the actions with more uncertainty, viz. less explored actions.

The algorithm uses the statistical mean, variance and an assumed distribution to calculate the confidence intervals. When the number of experiences is very large, a normal distribution is used. Otherwise, a t-distribution or a binomial distribution is used instead. The algorithm has been successfully used to solve n-armed bandit problems [69], robot goal keeper problems [81], and large maze problems [82] combined with prioritized sweeping [9].

It is worth noting that the algorithm is mainly designed to balance exploration and exploitation in stationary environments: more uncertain actions are explored more often. However, it can also be combined with non-greedy decision making methods or exploration bonuses to work in nonstationary environments. For instance, one extension of the algorithm [69] is to reduce the recorded number of visits to actions which have not been visited for some time. This, in effect, increases the upper interval boundary of these actions and therefore encourages exploration of them in a manner similar to exploration bonuses.

3.5.5 Bayesian methods

The environment where the agent is situated is usually unknown to the agent. The agent learns the environment through repeated interaction with the environment. It is not hard to understand that the more unknown the value of one action or one state-action pair is, the more the action or the state-action pair should be explored. A probability distribution can be used to model the uncertainty about the estimation of values. Take the n-armed bandit problem as an example. At the beginning of learning, Bayesian reinforcement learning methods assume a prior distribution, e.g. a uniform distribution or a normal distribution, over the probability of paying-off for each arm in a binary n-armed bandit problem and over the possible values of paying-off for each arm in a real-valued n-armed bandit problem. The probability distribution is then updated with experiences using Bayes' rule.

As for decision making, Wyatt [68] proposed to use a probability matching method to choose actions. At each step, the method chooses an action with the probability of the action being optimal. In this way, if the estimated returns of two arms are the same, the arm with a higher uncertainty is explored more often. With the probability distribution modelling the uncertainty about the expected return, it is possible to calculate the probability that an action is the optimal action. It is worth noting, however, when there are more than two actions, this method becomes intractable. In this case, a sampling method can be used [83]. With this method, a value is sampled from the estimated probability distribution of the value of each action. The action with the highest sampled value is chosen. The net result of the sampling method on average is the same with choosing one action with the probability of the action being optimal.

Although this method is not necessarily optimal for controlling the trade-off of exploration and exploitation, it is guaranteed to converge [68]. One drawback of the method is that it only considers the probability that an action is optimal, but ignores the amount by which choosing the action might improve over the current policy [83]. One exploration is more valuable if it can lead to bigger improvement over the current policy with a less or equal expected cost of taking a potentially suboptimal action. Following this thought, Dearden et al. [83] proposed to use Myopic-VPI (Myopic value of perfect information [84]) for decision making. Dearden et al. [83] further extended both the probability matching method and the Myopic-VPI method to multi-state reinforcement learning problems for Q learning.

For model based reinforcement learning, Dearden et al. [85] proposed to use Bayesian methods to learn the model of the environment. Assume that the agent takes action a in state s. One possible implementation is to use a parameter for each possible successor state s', viz. $\theta^{s'}_{s,a} = Pr(s'|s,a)$ to define the distribution of the transition, a parameter for each possible reward r, viz. $\theta^{r}_{s,a} = Pr(r|s,a)$ to define the distribution of the reward. Both $\theta^{s'}_{s,a}, s' \in S$ and $\theta^{r}_{s,a}, r \in R$ are categorical distributions. The number of times that s' is observed in n independent trials follows a discrete multinomial distribution with parameters $\theta^{s'}_{s,a}$ and n. The same is true with the reward distribution. In order to model the uncertainty about these parameters, we first assume a prior for each parameter, e.g. Dirichlet priors. Then we use Bayes' rule to calculate their posteriors after an experience is observed. With the Bayesian model of the environment, we can sample n times

of all the parameters and generate n MDPs. The value function of each MDPs can be obtained by dynamic programming or other reinforcement learning algorithms. The mean value function, viz. the average of all value functions, is then used together with either the probability matching method or the Myopic-VPI method for decision making.

It is worth noting that, similar to the interval estimation algorithm, these Bayesian reinforcement learning methods are mainly designed to model the uncertainty about the environment, and to balance exploration and exploitation in stationary environments rather than to solve problems arised due to nonstationary environments.

3.5.6 Risk sensitive reinforcement learning

The optimal target of classical reinforcement learning algorithms is to maximise the expected return either within a finite horizon or within an infinite horizon. It is not necessarily a desirable optimal target for all applications. Some applications may be risk averse while others may be risk seeking. Heger [86] introduced a reinforcement learning algorithm for the worst-case optimality criterion which chooses actions based on their worst-case returns instead of the expected return, viz. to maximise

$$\inf_{s_0,s_1,\ldots}(\sum_{k=0}^{\infty}\gamma^k r_{s_k,s_{k+1}}). \tag{3.22}$$

The algorithm usually results in a very low average return and therefore is only suitable for risk-avoiding applications.

Another approach is to use exponential utility functions. The goal is to maximise the expected value of an exponential utility function of the return. In this way, sample values are weighted depending on their distance away from the mean. One exponential utility function is $e^{\beta x}$, where the parameter β controls the desired sensitivity of risk. The goal is to maximise

$$\frac{1}{\beta}\, log\ E(e^{\beta\sum_{k=0}^{\infty}\gamma^k r_{s_k,s_{k+1}}}). \tag{3.23}$$

If we use the Taylor series to expand the equation, we can get

$$\frac{1}{\beta}\, log\ E(e^{\beta x}) = E(x) + \frac{\beta}{2}Var(x) + \mathcal{O}(\beta^2). \tag{3.24}$$

When $\beta = 0$, the agent only aims to maximise the expected return and therefore the objective becomes risk-neutral. When $\beta > 0$, variability, viz. risk, is encouraged and therefore the objective becomes risk-seeking. When $\beta < 0$, variability, viz. risk, is discouraged and therefore the objective becomes risk-averse. Thus, by setting β to different values, we can control the level of risk that we want the agent to take. This method, however, suffers from the following three problems [87]: time-dependent optimal policies, no optimality equations for stochastic reward structures, and no model-free reinforcement learning algorithms for both deterministic and stochastic reward structures. In order to avoid these drawbacks, Mihatsch and Neuneier [87] proposed to apply a utility function to the temporal differences instead of the return. A risk-sensitive version of Bellman equations is

$$0 = \sum_{s' \in S} p_{ss'}^{\pi(s)} \mathcal{U}(r_{ss'}^{\pi(s)} + \gamma V_\beta^\pi(s') - V_\beta^\pi(s)). \tag{3.25}$$

They further showed that, similar to risk neutral reinforcement learning, optimal stationary policies exist in the risk-sensitive sense and their optimal value function is unique and can be obtained by solving a risk-sensitive version of Bellman's optimality equations. In addition, most classical reinforcement learning algorithms can be transformed to their risk-sensitive version of algorithms easily. Risk-sensitive TD learning can be expressed as

$$V(s) \leftarrow V(s) + \alpha \mathcal{U}(r + \gamma V(s') - V(s)). \tag{3.26}$$

There are also risk-sensitive versions of other reinforcement learning algorithms, e.g. Q learning and SARSA learning. These risk-sensitive version of reinforcement learning algorithms converge under the same conditions with their normal (risk neutral) version.

It is worth noting that risk sensitive reinforcement learning allows the agent to take risk/variability into consideration for their decision making. It does not, however, address the problems caused by nonstationary environments.

3.5.7 Metacognitive monitoring and control

Metacognitive monitoring and control (MCL) [28,29,88,89] first detects the degree of perturbation in the environment. The degree of perturbation in the environment is estimated through monitoring the performance of the learning agent.

The performance indexes include the estimation of the expected reward value, the expected time to reward and the expected average reward per time step. For example, it would be considered a perturbation if the learning agent receives an unexpected reward, the number of time steps between rewards are three times the expected value, or the average reward per time step drops to eighty-five percent of the expected rate [29]. The degree of perturbation also depends on the severity of the perturbation, e.g. a reward change from 10 to -10 is a larger degree of perturbation than a reward change from 10 to 9. After a certain amount of perturbations have been detected, the method throws out its policy and starts over if the combined degree of perturbation is more than a threshold. Otherwise, it temporarily raises the exploration rate (e.g. ϵ for an ϵ-greedy policy) to encourage exploration and then allows the exploration rate to decay over time. The experiments by Anderson et al. [29] show that MCL has significantly improved the performance of Q learning, SARSA learning and Prioritized Sweeping when a perturbation happens, especially when it has a high degree.

Despite the fact that MCL has significantly improved the performance of classical reinforcement learning algorithms when the environment changes, it suffers from several drawbacks. Firstly, MCL calculates these performance indices only by averaging their actual values across several trials and then comparing the average value with their actual value in the current trial. Thus, it does not work in a stochastic stationary environment without using a batch rule, viz. comparing the average value across recent several trials with their average value across several trials before recent several trials. This is because the MCL approach would detect the variation in a stochastic stationary environment as a perturbation and therefore keep changing its policies. On the other hand, if a batch rule is used, it needs many trials to detect a change in the environment because it has to compare the average values across recent several trials with those across several trials before recent several trials.

Another disadvantage is that it only changes its policy between trials rather than during one trial. This would affect its response speed. As an extreme example, suppose, in an n-armed bandit problem, if the reward for one arm never arrives, their approach will still wait for the reward before changing its policy.

Thirdly, their method of calculating the degree of perturbation is rather heuristic and problem-dependent because all parameters are hand picked and problem-dependent. Furthermore, both the strategy of throwing out its policy

and the strategy of increasing the exploration rate after the detection of perturbation lack guidance because every state would be influenced. Finally, the method throws away the performance indexes and then relearns them from scratch after perturbations are detected rather than updates them. It takes some time before the performance indexes are learned and can be used to detect new perturbations.

3.5.8 Relational reinforcement learning

Almost all classical reinforcement learning algorithms use atomic states, viz. each value is associated with an atomic or an enumerated state. Instead of associating values with atomic states, relational reinforcement learning associates them with the relational representations of states described by first-order logical languages [90–92].

In doing so, it offers more generalisation which can not only reduce the state space but also make it more resilient to environmental changes. In a mobile rover problem, for instance, suppose that the environment has two kinds of terrains, viz. dry land and wetland, and also suppose that there is little difference between states with the same terrain. Suppose that there are two speed choices available to the mobile rover in both types of terrains. On a dry land, the mobile rover can receive more rewards by choosing the high speed, whereas it can receive more rewards by choosing the low speed on a wetland. Atomic reinforcement learning would associate values with the location of every state. Relational reinforcement learning, however, would associate values with the type of the terrain of every state. This way of state representation dramatically reduces the state space because, though there may be hundreds of distinct locations (hundreds of states for the atomic representations), there are only two kinds of terrains (two states for the relational representations). Now, assume that the terrain of a state changes from dry land to wetland. The learning agent with an atomic representation, would still use the same policy in the state that is used on a dry land (choosing to drive fast most of time) because it associates values with the location of the state and the location does not change. The learning agent with relational reinforcement learning, however, would use the policy on wetland in this state (choosing to drive slow most of time) instead because it associates values with the terrain of the state which has changed into wetland. Therefore, the performance of relational reinforcement learning has not been affected by the environmental change.

It is worth noting that this method is only useful when the relational representations of states are very simple. Suppose, in the above example, that if the value of a state depends not only on its terrain but also on the terrain of its neighbouring states, the relational representations would become more complex. In the extreme, if the value of a state also depends on its location, the relational representations would become as complex as the atomic representations. Secondly, relational reinforcement learning can only adapt to a change in the environment if the representation can characterise the change. Otherwise, it may still struggle to cope with the new environment. For example, if a state of the environment changes to a new type of terrain, e.g. grassland, relational reinforcement learning still needs to learn the change just like atomic reinforcement learning.

3.5.9 State augmentation

In a dynamic environment, if changes are decided by one or more dynamic elements, the problem can be transformed to a stationary environment problem by augmenting the state space with the dynamic elements [93, 94]. In a traffic control problem, for example, we can add a time index (e.g. peak time or off-peak time) to the state space and use different policies to control traffic in the peak time and off-peak time.

When there are too many dynamic elements, however, this method would cause an explosion of the state space. In a robot football problem, for instance, one can add all the positions of other footballers to the state space. Suppose there are 5 players per team and there are 100 possible positions, then the size of the new state space would become approximately 100^{10}.

3.5.10 State instantiation

One way to overcome the state space explosion problem suffered by the state augmentation method discussed above is to instantiate the states of dynamic elements in the environment model before learning, and use the instantiated model to learn new policies and then update the model with experiences [95].

To use this method, however, one has to know which objects are likely to change in advance because the cost of modelling a dynamic object is huge. Furthermore, like Dayan and Sejnowski's extended exploration bonus method discussed above, it also needs a model of the environment and assumes that the

environment changes with respect to a certain probabilistic distribution.

3.5.11 Methods designed specifically for cyclical environments

As mentioned previously, in a cyclical/recurrent environment, different types of environments with different reward structures and transitions between states appear repeatedly. In this case, it may be better to store the mapping of the type/mode of the environment and the policy/learning parameters (e.g. Q value for Q learning), and then recall the corresponding learning parameters when a type of environment reappears rather than relearning them from scratch.

Tsumori and Ozawa [76] store pairs of the policy/knowledge/parameters of the learning agent with the type of environment, and then recall the policy/knowledge corresponding to the type of environment which best matches the current environment when the environment changes. If no type of environment stored matches the current environment, the learning agent needs to learn it from scratch and then adds the pair of the learned policy and the type of the current environment to its knowledge base.

Choi et al. [75] integrate the state augmentation method with a hidden Markov model. Here, the mode of the environment is the hidden variable of the hidden Markov model and the current state is the observation. The learning agent first learns a policy in each mode of the environment, and then associates the policy with the corresponding mode and stores the association. It switches to an appropriate policy according to the current mode of the environment deduced from the current state (observation) when the environment changes. However, it assumes that the approximate number of different environment dynamics (modes) is known. Silva et al. [96] lift this limitation by incrementally building new models if the current mode is new.

It is worth noting that these methods only work in a cyclical/recurrent world where different reward structures and transitions between states appear repeatedly.

3.6 Our research

Classical value estimation reinforcement learning algorithms can learn very quickly but cannot respond rapidly to abrupt changes in the environment. Exploration bonuses improve the speed of response to abrupt changes in the environment at the price of the learning speed because it always explores the places which have not been visited recently whether the environment has changed or not. In order for the agent to both learn quickly and respond rapidly to abrupt changes in the environment, we propose to learn and monitor the time to reward for every state-action pair. The learned time information is then used to detect changes in the environment. If a change in the environment is detected, the learning rate is increased in order for the agent to learn the change quickly. Otherwise, the learning rate is decreased gradually towards 0 in order for the estimated mean of the time to reward to converge to its true mean. If a suboptimal action has improved and may potentially become the optimal action in the new environment, the exploration rate (e.g. ϵ for ϵ-greedy) is also increased in order to increase the chance that the suboptimal action is visited. Otherwise, the exploration rate is decreased gradually towards its minimum value in order to reduce the cost of exploring suboptimal actions. In addition, when the learning agent has found that the current action is still worse than the optimal one some time after taking the action, as discussed previously, it gives up this time's exploration of the action in order to avoid longer than necessary exploration.

The MCL approach is similar to our research in the sense that it also detects changes in the environment through monitoring the performance of the learning agent and then changes its policy if a change in the environment has been detected. However, both its detection method and its response strategy are rather heuristic because parameters are hand-picked and problem dependent (e.g. the expected reward is not received three times in a row). In contrast, our algorithm detects changes in the environment with the mean and variance of the time to reward which is suitable for almost all environments. Furthermore, their method also lacks guidance because every state would be influenced if a change in the environment is detected, e.g. throwing out its policy and temporarily raising the exploration rate for all states. In contrast, our algorithm only increases the learning rate of the state-action pair and ϵ of the state whose time to reward or value has changed after a change in the environment is detected.

Secondly, MCL only calculates the performance indexes and changes its policy

between trials, whereas our algorithm can respond (e.g. give up the current action) during one trial. In doing so, our algorithm can potentially respond to changes in the environment more quickly. Suppose, in an n-armed bandit problem, if the reward for one arm never arrives, their approach will still wait for the reward before changing its policy which will never happen. Our algorithm, however, can detect the problem and then give up the arm and try other arms instead.

Thirdly, MCL only calculates the mean of the time to reward, so it would not work without using a batch rule in a stochastic stationary environment where the value of reward or the time to reward for state-action pairs keeps changing. This is because the MCL approach would detect the variation in a stochastic stationary environment as a perturbation and therefore keep changing its policy. On the other hand, if a batch rule is used, it needs many trials to detect a change in the environment because it has to compare the average values across recent several trials with those across several trials before recent several trials. Our algorithm learns both the mean and variance of the time to reward, so it can detect a change in the environment with only one trial whether in deterministic or stochastic environments. Finally, MCL throws away the performance indexes and relearns them after perturbations are detected whereas our algorithm updates and reuses them.

Chapter 4

Classical conditioning with spiking neurons

As mentioned previously, classical conditioning [53] can be found in almost all living organisms and may be one of the most important learning mechanisms behind the independent learning behaviours of organisms. Some robotics researchers [97] believe that implementing classical conditioning in robots would be the first step to build a robot that can learn independently like animals. Classical conditioning is also a simple form of reinforcement learning and a motivation for temporal-difference (TD) learning, an important reinforcement learning algorithm. Traditionally, classical conditioning is mainly modelled by connectionist neural networks, e.g. the Rescorla-Wagner model [36] and the Sutton-Barto model [37]. These models use high-level abstractions of neurons which ignore the temporal dynamics of real neurons. Classical conditioning, however, is dynamic in essence because it mainly learns and predicts dynamic environments. The Sutton-Barto model [37] solves this problem by using a temporal learning rule. Here we attempt to use the dynamics of biological neurons directly to model the dynamic nature of classical conditioning as the first step of our research on reinforcement learning with time perception.

Experiments by Rao and Sejnowski [98] show that the temporally asymmetric window of Hebbian plasticity can be reproduced by a temporal-difference (TD) learning rule in conjunction with dendritic backpropagating action potentials. This chapter explores the possibility of the inverse problem, viz. is it possible to use spike-time dependent Hebbian learning and spiking neuron models to realise TD learning and further to model classical conditioning?

The first section provides some preliminary knowledge of spiking neuron models. The neuron model and learning algorithm, used to implement classical conditioning in our experiments, are introduced respectively in section 4.2 and section 4.3. Then, the results of the simulation are shown in section 4.4. Finally, the last section concludes the chapter by discussing the robustness of the model and other related topics.

4.1 Background on spiking neuron models

Traditionally, most models of neural networks, whether in artificial neural networks or in neuroscience, assume that neurons encode information by means of their firing rates. Typically, these models take a large number of neuron-like processing units, connect them together with weighted connections that are the rough equivalent of neural synapses, and assume the output of each unit is some activation function or transfer function, which mimics the dynamics of neurons, of the weighted sum of all the inputs to each neuron. Besides, they also take it for granted that signals transmitted in the networks (viz. inputs and outputs of each unit) are a continuous value, often a floating point number. This, however, is not the case for biological neurons. In the biological neural system, neurons communicate by means of a sequence of short electrical pulses, the so-called spikes or action potentials rather than a continuous value. Furthermore, data gathered in recent years from neurobiological experiments [99–101] also support that information is transmitted through spikes and the timing of these spikes is used to transmit information and perform computation. This realisation has stimulated a significant growth of research activity in the area of spiking neural networks or pulsed neural networks [102].

We shall give a short introduction to the biological neural system in subsection 4.1.1. In subsection 4.1.2, we will present two different information coding schemes which can be used by neurons and discuss their advantages and disadvantages. The spiking neuron models are mainly focused on in subsection 4.1.3 for single neurons and in subsection 4.1.4 for population neurons. In subsection 4.1.5, different synapse models are introduced. The last subsection discusses different learning algorithms which can be used by neurons.

4.1.1 The biological neural system

The biological neural system is comprised by billions of neurons connected with each other through synapses. Typically, a neuron is composed by dendrites, cell body and axon.

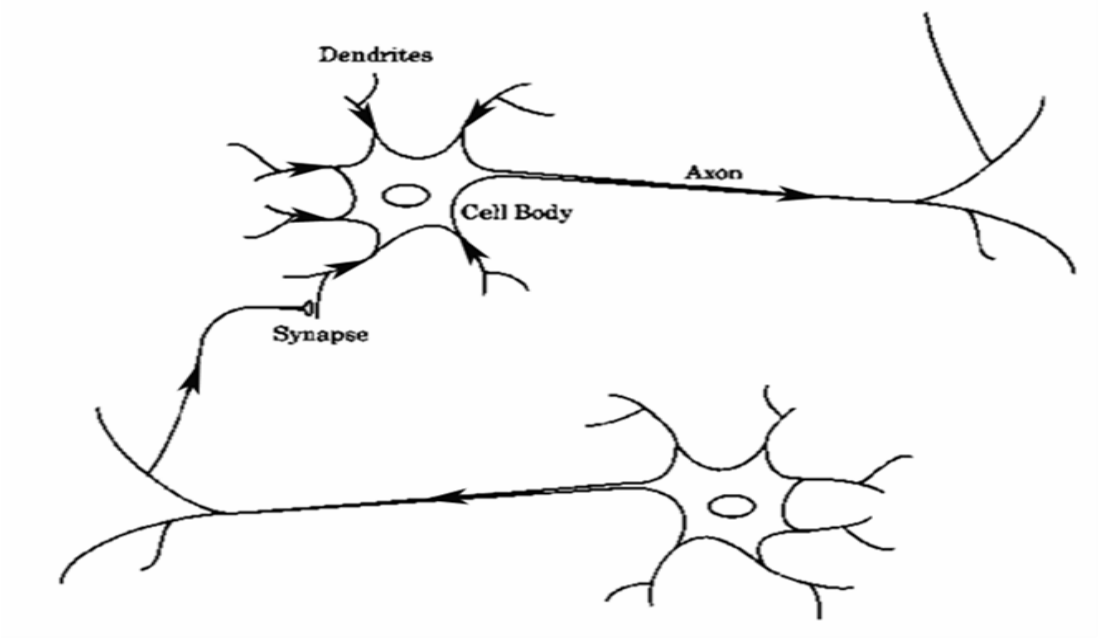

Figure 4.1: Schematic illustration of biological neurons (taken from [103]).

As figure 4.1 shows, signals (information about the environment) are transmitted from the axon of one neuron to the dendrite of another neuron through a synapse. When across synapses, signals may be amplified, reduced, or delayed. In the cell body of the neuron, signals from different neurons will then be computed. Next, the computed signal will be sent through the axon of the neuron to other neurons connected with it.

4.1.2 Neural coding

The brain is well known as an information processing system. Information about the environment is received through receptors, such as eyes, nose, ears, skin and taste buds, and then is sent to the brain for processing. After processing the information, the brain will send control signals to effectors, such as the muscles of hands and feet, to respond to the environment. Finally, after receiving the control signals from the brain, effectors will act to adapt to the environment.

The problem of neural coding is to find how information from the environment is encoded by receptor neurons, how the coded information is decoded by the brain, how the brain encodes the action signals, and how effector neurons decode the coded control information.

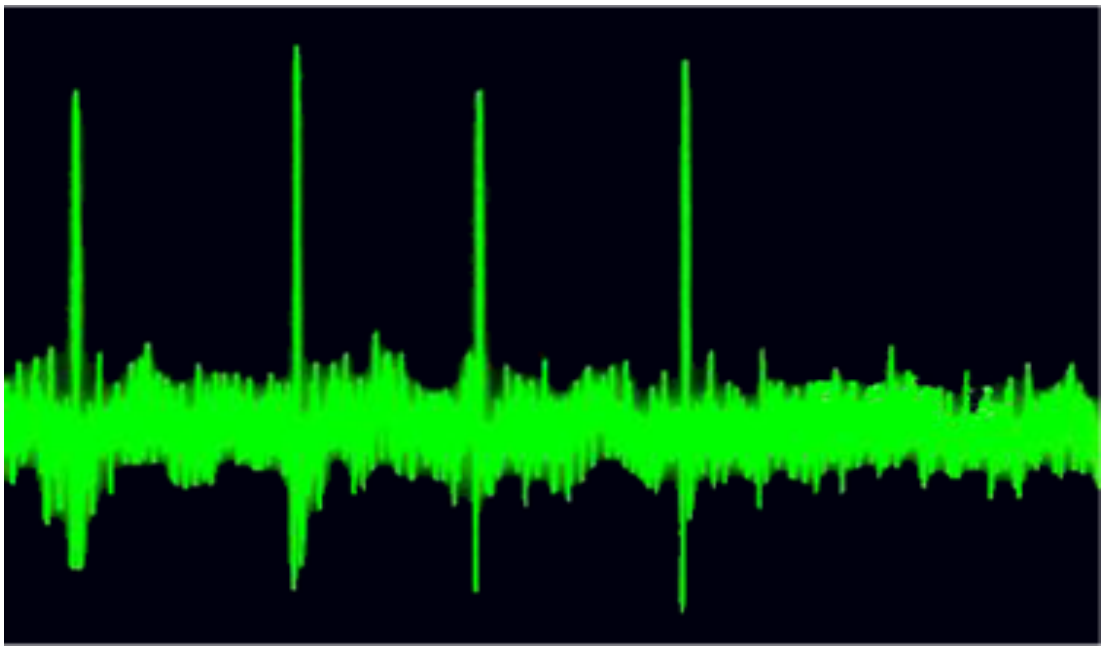

Figure 4.2: Signals generated and transmitted by neurons(taken from [101]).

As figure 4.2 presents, signals generated and transmitted by neurons are just short electrical pulses or action potentials or spikes, not waves or any other forms. There are four spikes in this figure and others are background noise. From the figure, we can also see that there is little difference between the shapes of spikes. Thus, it is likely that neurons encode information either through the amount of spikes (the firing rate model) or through the timing of spikes (the spiking neuron model), both of which will be discussed below.

4.1.2.1 Firing rate models

The firing rate model has been used since the very beginning of experimental neurophysiology. In 1920s, the pioneering work of Adrian [104, 105] suggested that the firing rate of stretch receptor neurons in muscles is related to the force applied to the muscles. Since then, the firing rate model has become the dominant model used in both artificial neural networks and neuroscience partly because of the relative ease of measuring firing rates experimentally.

According to different averaging procedures, the firing rate model can be categorised into three kinds, viz. temporal average coding, repetition average coding and population average coding.

The temporal average coding is the most commonly used one of the three categories. It is defined as an average of the number of spikes over a period of time, usually 100 ms or 500 ms. If one sets the time window T and counts the number of spikes n_{sp} occurred from t to $t+T$, according to Gerstner and Kistler [102], the firing rate v at t for this coding is

$$v(t) = \frac{n(t, t+T)}{T}. \tag{4.1}$$

Although the temporal average coding is relatively simple and has been successfully used by experimenters to evaluate and classify neuronal firing, it is not suitable for neurons in the biological neural system to encode information. In the real world, creatures have to react to stimuli very quickly to stay alive. For instance, a fly can react to a stimulus very quickly and change the direction of its flight within 30–40 ms [101]. This means that the fly has to respond after a post-synaptic neuron has received just one or two spikes. So, there is not enough time for the fly to count spikes and average them over a long time window. Primates can also respond selectively to complex visual stimuli such as faces, food and other familiar 3D objects only 100–150 ms after stimulus onset [99]. Moreover, there are more than 10 layers of neurons on the way from the retinal photoreceptors to the brain. So, each layer of neurons can only have about 10 ms to process information. Experiments on bats [100] also show that neurons in the bat auditory cortex can respond just 8 ms after stimulus onset, which leaves only a couple of milliseconds at each level given the number of intervening subcortical processing stages. Thus, even though it is convenient for experimenters to evaluate and analyse neuronal firing using the temporal average coding, it cannot be the coding that the biological neural system actually uses.

In the repetition average coding, the same stimulation sequence is repeated several times and then the neuronal responses are averaged over these repetitions. Suppose, at the time t, a neuron produces $n_K(t, t+\Delta t)$ spikes summed over K repetitions of the same experiment in a very short time period of Δt. Then, according to Gerstner and Kistler [102], the firing rate of the neuron at the time

t for the repetition average coding is

$$v(t) = \frac{1}{\Delta t}\frac{n_K(t, t+\Delta t)}{K}. \tag{4.2}$$

Although the repetition average coding is useful to evaluate and analyse neuronal activity, it is apparent that the biological neural system cannot use it to encode information either. This is because neurons must wait for several repetitions of stimuli before they can get the firing rate with the repetition average coding, which is apparently unrealistic in the real world.

Among numerous neurons, some of them have similar properties and respond to the same stimuli. So, it is reasonable to average the activity of these neurons to get a more accurate description of neuronal activity. Assume there are n_N spikes produced by N neurons with similar properties from the time t to the time $t+\Delta t$ (Δt is a very short time period). Then, according to Gerstner and Kistler [102], the firing rate of the neurons for the population average coding at the time t is

$$v(t) = \frac{1}{\Delta t}\frac{n_N(t, t+\Delta t)}{N}. \tag{4.3}$$

4.1.2.2 Spiking neuron models

Because of the limitations of firing rate models, researchers gradually realised the importance of the timing of spikes and increasingly started research in spiking neuron models. A number of spiking neuron models are introduced. Here, we only discuss three types of them, viz. rank order coding, temporal coding and population correlation coding.

Rank order coding [106–109], the simplest one of the three coding schemes, only considers the order in which neurons fire instead of the exact timing of spikes produced by neurons. The basis for this coding scheme is that neurons only have time to fire either 0 or 1 spike for rapid processing. Therefore, the first spike should contain most of the relevant information.

Unlike rank order coding, temporal coding makes use of the precise timing of all spikes produced by neurons. Without doubt, this coding scheme is potentially powerful and can contain a large amount of information.

Neurons in the cortex are connected with each other and therefore interact with each other. So, their activity is usually correlated and this kind of correlation may provide important information that is not contained in the timing

of spikes produced by individual neurons. For this reason, it may be groups of correlated neurons that together encode information instead of isolated neurons. For example, the brain may use synchrony (a special case of correlation without time lag, viz. two or more neurons fire at the same time) to tell whether or not spikes produced by different neurons belong to the same visual object.

4.1.3 Single neuron models

In the paradigm of spiking neuron models, there are mainly two kinds of single neuron models, one is detailed conductance-based neuron models and the other simple phenomenological spiking neuron models. Because of the intrinsic complexity of detailed conductance-based neuron models, only simple phenomenological spiking neuron models are discussed here.

According to how to define the dynamics of the membrane potential of neurons, the simple phenomenological spiking neuron models can be classified into several groups. Here, we only consider two of them, viz. the leaky integrate-and-fire model and the spike response model.

4.1.3.1 Leaky integrate-and-fire model

The leaky integrate-and-fire model uses a basic RC circuit showed in Figure 4.3 to simulate the dynamics of neuronal activity. When input spikes arrive at a neuron, the membrane potential of the neuron is integrated. The neuron will fire a spike when the membrane potential reaches some value, viz. its threshold. On the other hand, the membrane potential of the neuron will leak over time and may even disappear if there are no input spikes.

As figure 4.3 represents, the circuit is composed by a capacitor C and a resistor R. A current $I(t)$ drives the circuit and is split into two components I_R and I_C. If the capacity of C at time t is denoted as $u(t)$, then, according to Gerstner and Kistler [102], $I(t)$ can be expressed as

$$I(t) = I_R + I_C = \frac{u(t)}{R} + C\frac{du(t)}{dt}. \tag{4.4}$$

Let $\tau_m = RC$, then, according to Gerstner and Kistler [102], equation 4.4 can be represented as

$$\tau_m \frac{du(t)}{dt} = -u(t) + RI(t). \tag{4.5}$$

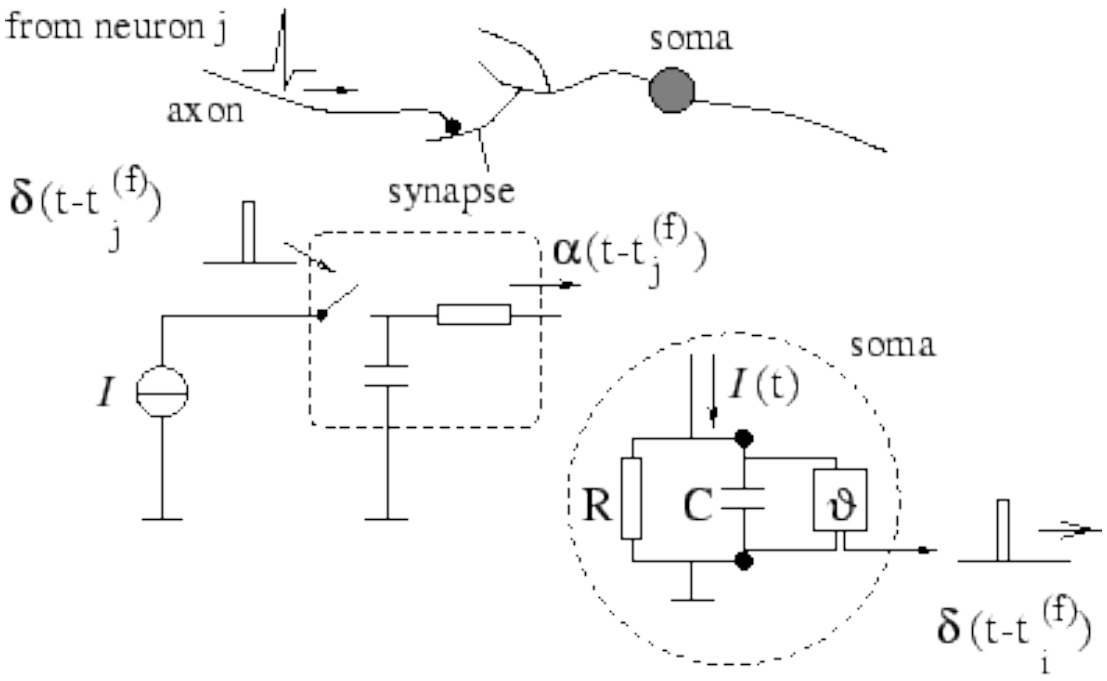

Figure 4.3: Schematic diagram of the leaky integrate-and-fire model (taken from [102]).

In terms of the leaky integrate-and-fire model, $u(t)$ refers to the membrane potential of the postsynaptic neuron, τ_m refers to the membrane time constant of the neuron, $I(t)$ refers to spikes of presynaptic neurons, and R measures the influence of external currents on the membrane potential.

Equation 4.4 and equation 4.5 only describe the neuronal activity between the firing of one spike and the firing of the next spike by the postsynaptic neuron, viz. before the the membrane potential of the postsynaptic neuron reaches its firing threshold. According to Gerstner and Kistler [102], when the membrane potential $u(t)$ of the postsynaptic neuron reaches its firing threshold υ at time $\hat{t}$, viz.,

$$\hat{t} : u(\hat{t}) = \upsilon, \tag{4.6}$$

a spike will be fired by the postsynaptic neuron. And immediately after the firing moment, its membrane potential is reset to $u_r < \upsilon$. Then, the dynamics of the postsynaptic neuron will develop along the trajectory described by equation 4.4 and equation 4.5 until its next spike.

4.1.3.2 Spike response model (SRM)

Unlike the leaky integrate-and-fire model, the spike response model (SRM) assumes a function ϵ to describe the time course of the response of the membrane potential of a neuron to an incoming spike instead of a basic RC circuit and a function η to describe the time course of the membrane potential of a neuron after firing a spike instead of simply resetting the membrane potential of the neuron to u_r.

Suppose that the postsynaptic neuron i fired its last spike at time $\hat{t}_i$. After that, at time $t_j^{(f)}$, the presynaptic neuron j generated a spike which caused the membrane potential (u_i) of the neuron i to increase by $w_{ij}\epsilon_{ij}(t-\hat{t}_i, t-t_j^{(f)})$ at time t. f means firing and $t_j^{(f)}$ is the time when the presynaptic neuron j fires a spike. Besides, the trajectory of the membrane potential after firing a spike can be described by the same certain time course function $\eta(t-\hat{t}_i)$ because action potentials always have roughly the same form. So, the membrane potential (u_i) of the neuron i at time t, according to Gerstner and Kistler [102], can be expressed as

$$u_i(t) = \eta(t-\hat{t}_i) + \sum_j w_{ij} \sum_f \epsilon_{ij}(t-\hat{t}_i, t-t_j^{(f)}). \tag{4.7}$$

4.1.4 Population neuron models

The biological neural network is an inherent parallel processing system in which neurons are highly connected with each other (approximately 7,000 connections per neuron on average [103]). Thus, it is not appropriate to isolate one neuron from others and only consider the models of a single neuron. Furthermore, there are a large number of neurons that are located nearby, have similar properties and perform a certain function together in the brain. For instance, there are the somatosensory cortex within which a large number of neurons perform the somatosensory function [110], the visual cortex within which a large number of neurons perform the visual function [111], and pools of motor neurons within which a large number of neurons perform the motor function [112]. Therefore, it is essential to consider the activity of the whole population of neurons rather than that of individual neurons.

Because of the restriction of length and the complexity of population neuron models, however, we will not further discuss the population neuron model.

4.1.5 Synapses

The synapse is where the axon of a presynaptic neuron makes contact with the dendrite or soma of a postsynaptic neuron. As mentioned previously, signals (or spikes in spiking neuron models) are transmitted from the axon of one neuron to the dendrite of another neuron through synapses. When they cross a synapse, signals may be amplified, attenuated, or delayed. The degree to which the signals are amplified or attenuated depends on the strength of the synapse which is also plastic. The synapse is believed to be the foundation of learning and memory.

An α-function [102] is usually used to model the synapse. There are two versions of α-function. The simple version has only one parameter. Suppose $I(t)$ is the current triggered by spikes from other neurons or external sources at time t, which is also the current that charges the membrane potential of the postsynaptic neuron, then, according to Gerstner and Kistler [102], it can be expressed as

$$I(t) = \frac{1}{\tau_{\mathrm{s}}} \sum_j \sum_k w_j e^{-\frac{t-t_j^k-d_j}{\tau_{\mathrm{s}}}} \theta(t - t_j^k - d_j) \tag{4.8}$$

where, τ_{s} is the time constant, w_j and d_j are respectively the synaptic weight and the transmission delay of the jth connection, and t_j^k is the kth input spike time from the jth synapse. $\theta(s)$ is the Heaviside step function and satisfies,

$$\theta(s) = \begin{cases} 0 & \text{if } s \leq 0 \\ 1 & \text{if } s > 0 \end{cases}$$

A more sophisticated version has two parameters. In addition to τ_{s}, the other parameter is the finite rise time τ_{r} of the postsynaptic current. This version of α-function, according to Gerstner and Kistler [102], can be expressed as

$$I(t) = \frac{1}{\tau_{\mathrm{s}} - \tau_r} \sum_j \sum_k w_j (e^{-\frac{t-t_j^k-d_j}{\tau_{\mathrm{s}}}} - e^{-\frac{t-t_j^k-d_j}{\tau_{\mathrm{r}}}}) \theta(t - t_j^k - d_j). \tag{4.9}$$

4.1.6 Neural learning

Only a fraction of neural structures are defined at birth [103]. Most parts are developed with new connections made and others waste away in the early stage of life to learn the environment. After that, they continue changing throughout the whole life to adapt to changes in the environment, but these changes are mainly

limited to the strengthening or weakening of synaptic connections. Thus, neural learning is very important for creatures to fit the environment, stay alive and flourish.

According to whether there is external information that helps a learning agent to learn and whether the external information is instructive or evaluative, the learning process can be grouped into three categories, viz. supervised learning, unsupervised learning and reinforcement learning.

Recent research [113, 114] indicates that the biological neural system involves all of the three learning paradigms. Specifically, it is indicated that the cerebral cortex is specialized for unsupervised learning, the cerebellum for supervised learning, and the basal ganglia for reinforcement learning.

4.1.6.1 Unsupervised learning

Unlike supervised learning, unsupervised learning has no teacher to oversee its learning process. Therefore, it can only learn from inputs.

Unsupervised learning in artificial neural networks can be realised by associative learning [115, 116], competitive networks [117] and Grossberg network [118]. Their main application is to categorise the input patterns into a finite number of classes.

Unsupervised learning for spiking neuron models is usually realised by time-dependent Hebbian learning and maximum mutual information theory.

The basic rule of Hebbian learning is from Hebb's postulate,

> "When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that A's efficiency, as one of the cells firing B, is increased."

Although Hebb never claimed that there is firm physiological evidence for his theory, subsequent research has shown that neurons in this case do exhibit this kind of learning. Recent experiments [119–133] push Hebbian learning further and suggest that Hebbian learning is time-dependent, viz. if a presynaptic spike arrives at the synapse before the postsynaptic action potential, the synapse is potentiated (long-term potentiation (LTP)); if the timing is reversed the synapse is depressed (long-term depression (LTD)). This kind of time-dependent Hebbian learning is also called spike-timing dependent plasticity (STDP) [132, 134, 135].

Moreover, the curve of time-dependent anti-Hebbian learning is also observed in the cerebellum of electric fish [124, 131], viz. if a presynaptic spike arrives at the synapse before the postsynaptic action potential, the synapse is depressed; if the timing is reversed the synapse is potentiated.

Besides, other researchers [136–138] are concerned with designing an optimal synaptic updating rule so as to maximise the mutual information between pre- and postsynaptic neurons. They found that the learning rule produces a learning window similar to that of spike-timing dependent plasticity (STDP).

4.1.6.2 Supervised learning

Supervised learning is learning from an external teacher [139]. A typical scenario of supervised learning is that a learning agent tries to learn the environment where it exits. After receiving inputs (the current state of the environment, for example) from the environment, the agent will give an estimate based on its internal parameters. At the same time, a teacher who has full knowledge of the environment will tell the agent the correct answer to the input. Then, the agent will adjust its internal parameters to minimise the errors between its estimation and the teacher's answer. After many times' corrections, the agent will be able to predict the environment correctly.

For artificial neural networks with only a single layer, so called the Perceptron [140, 141], supervised learning can be realised by using the least mean square (LMS) or delta rule [142, 143]. At the k iteration, suppose, $\boldsymbol{p}(k)$ is the input vector of a neural network, $\boldsymbol{W}(k)$ is the weight matrix of the input vector to the neural network, $\boldsymbol{b}(k)$ is the bias vector of the neural network, $\boldsymbol{e}(k)$ is the error vector between the actual output vector $\boldsymbol{a}(k)$ and the target output vector $\boldsymbol{t}(k)$, and $\alpha(k)$ is the learning rate. In this case, the LMS rule can be expressed as

$$\boldsymbol{W}(k+1) = \boldsymbol{W}(k) + 2\alpha(k)\boldsymbol{e}(k)\boldsymbol{p}^T(k). \tag{4.10}$$

For artificial neural networks with multiple layers, the backpropagation algorithm [42, 144, 145] is used to solve the problem of the credit or blame assignment problem (how to propagate the errors back to the layers preceding the last layer so that neurons in these layers can also learn the errors).

After the popularisation of spiking neuron models, great efforts are made to implement supervised learning using spiking neuron models. Legenstein, Naeger

and Maass [146] examined what a neuron can learn with spike-timing dependent plasticity in the supervised learning paradigm. Their experiments showed that the convergence of the supervised learning from arbitrary inputs to output spike patterns can not be guaranteed in contrast to the perceptron convergence theorem. On the other hand, however, they also proved that the convergence can be achieved for both uncorrelated and correlated Poisson input spike trains in an average case sense.

Barber et al. [147–150] designed a supervised learning rule to maximise the likelihood of postsynaptic firing at one or several desired firing times. It was found that the learning window of the optimal strategy is similar to that of spike-timing dependent plasticity.

4.1.6.3 Reinforcement learning

Like unsupervised learning, reinforcement learning also has no teacher to give the correct answer; but unlike unsupervised learning, it does have a critic who will give a reward or penalty to show how good it behaves after it takes an action. Although the learning agent can get feedback from the environment or a critic in reinforcement learning, reinforcement learning is different from supervised learning in that the learning agent is not told what it should have done.

Reinforcement learning is a more plausible learning for creatures than supervised learning because creatures generally have no teachers to tell them what they should do. On the other hand, however, they do receive rewards (e.g. food, life and happiness) and penalty (e.g. hunger, death and pain) after their actions. Studies [113] suggest that the basal ganglia of the brain may be involved in reinforcement learning. More specifically, studies by Schultz et al. [58, 59] indicate that the activity of the dopaminergic neurons in the basal ganglia may be used to encode the error between the actual reward and the expected reward. Studies by Barto [151], Houk and et. al. [152] indicate that the striosome compartment of the striatum in the basal ganglia may be used to predict the value of reward and the matix compartment of the striatum in the basal ganglia may be used to make action selection. In addtion, experiments by Rao and Sejnowski [98] show that the temporally asymmetric window of spike-timing dependent plasticity can be reproduced by a reinforcement learning rule in conjunction with dendritic backpropagating action potentials.

4.2 Neural structure

4.2.1 Neuron architecture

Our model uses only one neuron and regards both unconditional stimulus (US) and conditional stimuli (CS) as inputs of the neuron as shown in figure 4.4. US is regarded as a special stimulus with a big fixed weight (set to 1 in the simulation). The output of the neuron is an unconditional response (UR)/ a conditional response (CR) depending on whether it is produced by US or CS. All of these inputs and outputs are expressed in the form of spikes.

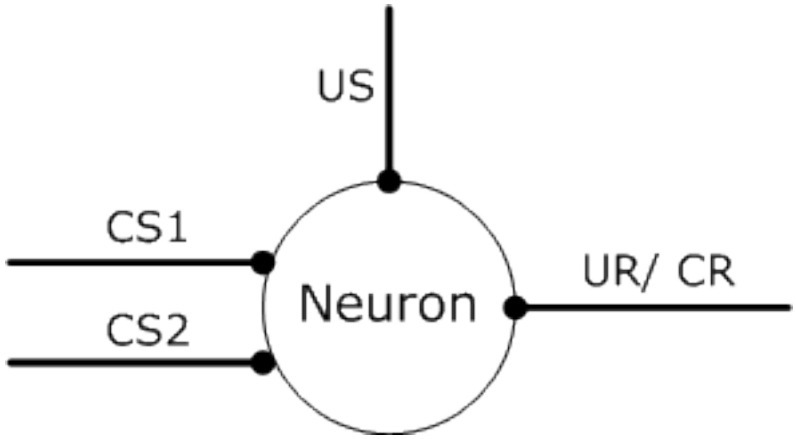

Figure 4.4: Neuron architecture used to model classical conditioning. Both unconditional stimulus (US) and conditional stimuli (CS) are inputs of the neuron. US is regarded as a special stimulus with a big fixed weight (set to 1 in the simulation). The output of the neuron is an unconditional response (UR)/ a conditional response (CR) depending on whether it is produced by US or CS. All of these inputs and outputs are expressed in the form of spikes.

4.2.2 Neuron model

The leaky linear integrate-and-fire model introduced in section 4.1.3.1 is used to represent the dynamics of the neuron in the simulation for simplicity.

In the numerical simulation, the parameters of the neuron model are set as follows: $\tau_{\mathrm{m}} = 10\mathrm{ms}$, $R = 10\mathrm{k}\Omega$, $v = 1\mathrm{mV}$, $u_{\mathrm{r}} = 0\mathrm{mV}$.

4.2.3 Synapse model

The simple version of the α-function introduced in section 4.1.5 is used in the simulation to model the neuron's synapses.

In the numerical experiment, we set $\tau_{\mathrm{s}} = 10\mathrm{msec}$, $d_j = 0\mathrm{msec}$.

4.3 Learning algorithm

Although spike-time dependent Hebbian learning is well-known in the biological neural system, the curve of spike-time dependent anti-Hebbian learning is also observed in the cerebellum of electric fish [124, 131], viz. if a presynaptic spike arrives at the synapse before the postsynaptic action potential, the synapse is depressed; if the timing is reversed the synapse is potentiated.

It has also been found that long-term potentiation (LTP) [153] of the cortico-striatal synapse is induced given the coincident occurrence of a cortical input and postsynaptic depolarization in a striatal neuron with a phasic dopamine release, and long-term depression (LTD) [154] given the coincident occurrence of a cortical input and postsynaptic depolarization in a striatal neuron without a phasic dopamine release. Since it is generally believed that dopamine is involved in reward learning, this suggests that Hebbian learning happens when rewards are present and anti-Hebbian learning happens when rewards are not present.

Furthermore, in almost all the experiments [122,123,126,127] where spike-time dependent Hebbian learning is found, the action potentials in the postsynaptic neuron are initiated by current injection rather than the presynaptic spikes. Therefore, spike-time dependent Hebbian learning may be formed between the presynaptic spikes and the postsynaptic action potentials produced by other inputs (the current injection) rather than the action potentials produced by the presynaptic spikes themselves. Besides, it is also possible that spike-time dependent Hebbian learning is formed between the presynaptic spikes and the current injection (which can be considered as an unconditional stimulus or a reward) instead of the postsynaptic action potentials.

Based on these findings, spike-time dependent Hebbian learning is applied between inputs and spike-time dependent anti-Hebbian learning is applied between the input and the output in our model. For simplicity, a square learning window is used as shown in figure 4.5 to mimic the curve of spike-time dependent Hebbian learning and spike-time dependent anti-Hebbian learning. US has a big

fixed weight affected neither by CS nor by UR/CR. CS has a small weight at the beginning but it can be either potentiated or depressed by US, UR/CR and other CS.

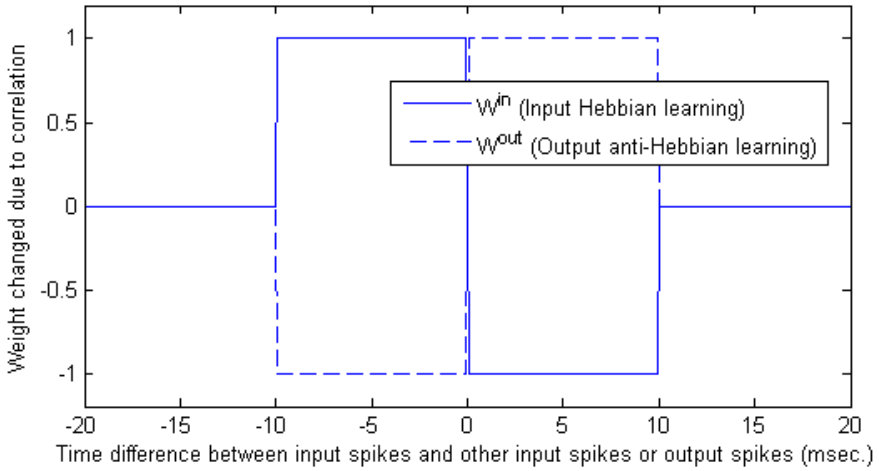

Figure 4.5: Square learning window. As the solid line shows, the weight associated with an input (excluding US) will be reinforced if a spike of the input is followed by spikes of other inputs; it will be weakened if a spike of the input is preceded by spikes of other inputs. As the dashed line shows, the weight associated with an input (excluding US) will be weakened if a spike of the input is followed by spikes of the output; it will be reinforced if a spike of the input is preceded by spikes of the output.

Suppose there are N stimuli including both CS and US. W^{in} and W^{out} respectively denote the learning window of input Hebbian learning and that of output anti-Hebbian learning respectively as figure 4.5, equation 4.11 and 4.12 show.

$$W^{\text{in}}(t) = \begin{cases} 1 & \text{if } -\tau < t < 0 \\ -1 & \text{if } 0 < t < \tau \\ 0 & \text{otherwise} \end{cases} \tag{4.11}$$

$$W^{\text{out}}(t) = \begin{cases} -1 & \text{if } -\tau < t < 0 \\ 1 & \text{if } 0 < t < \tau \\ 0 & \text{otherwise} \end{cases} \tag{4.12}$$

where τ is the size of the learning window and $\tau = 10msec.$ in the experiments.

The weight increment of the ith stimulus between time t and time $t+T$ (T is a period of time) is denoted as $\Delta w_i(t, t+T)$. Then, the mathematical formula of

our learning algorithm can be expressed as equation 4.13 for CS and equation 4.14 for US. If the ith stimulus is a CS, then

$$\Delta w_i(t, t+T) = \sum_{j=1 \cap j \neq i}^{N} \alpha \int_t^{t+T} dt' \int_t^{t+T} dt'' w_j(t) W^{\text{in}}(t''-t') S_i^{\text{in}}(t'') S_j^{\text{in}}(t') + \alpha \int_t^{t+T} dt' \int_t^{t+T} dt'' W^{\text{out}}(t''-t') S_i^{\text{in}}(t'') S^{\text{out}}(t'). \tag{4.13}$$

The first term is the weight update due to the Hebbian learning effect of other inputs as shown by the solid line in figure 4.5, and the second term is the weight update due to the anti-Hebbian learning effect of the output as shown by the dashed line in figure 4.5. α is the learning rate (set to 0.01 in the simulation), and

$$S^{\text{in}}(t) = \begin{cases} 1 & \text{if } t = \hat{t} \text{ ($\hat{t}$ is the arrival time of spikes of inputs)} \\ 0 & \text{otherwise} \end{cases}$$

$$S^{\text{out}}(t) = \begin{cases} 1 & \text{if } u(t) \geq \upsilon \\ 0 & \text{otherwise} \end{cases}$$

where $u(t)$ is the membrane potential and υ is the firing threshold of the neuron.

If the ith stimulus is a US, then

$$\Delta w_i(t, t+T) = 0. \tag{4.14}$$

4.4 Simulation results

Although we have successfully simulated a variety of classical conditioning phenomena, we only discuss the four most important of them here, viz. Pavlovian conditioning, extinction, blocking and secondary conditioning.

4.4.1 Pavlovian conditioning

In the setting of the Pavlovian conditioning experiment, CS1 has 15 spikes from the 16th millisecond to the 30th millisecond, followed by US which also has 15 spikes from the 31st millisecond to the 45th millisecond, and no CS2 is presented

as figure 4.6a & b depict. Before learning, UR, induced by US, is shown in figure 4.6c; the initial weights respectively associated with CS1 and associated with CS2 are both set to 0. During learning, the weight associated with CS1 keeps increasing until it reaches a saturation value near 0.7 and the weight associated with CS2 remains unchanged as Figure 4.7 presents. After learning, CS1 alone can produce output spikes (CR which precedes UR and predicts the approach of US) as shown in figure 4.6d. Pavlovian conditioning has been formed.

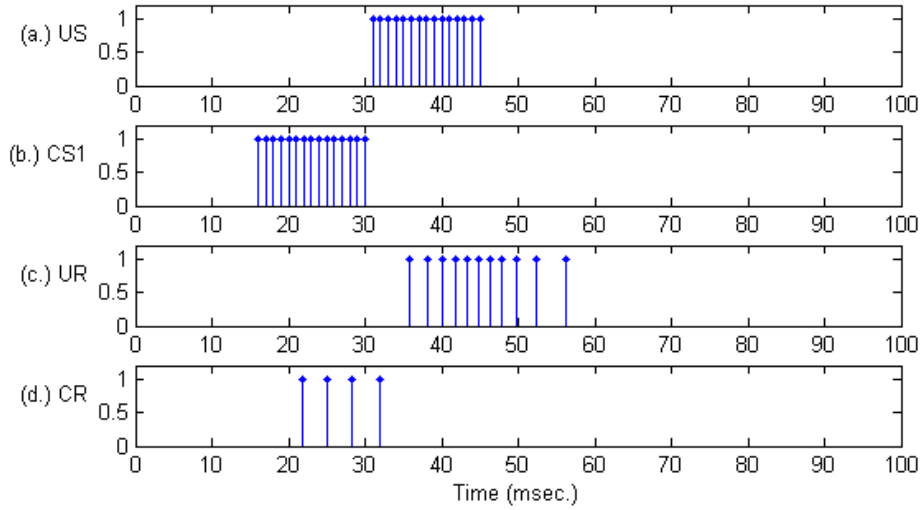

Figure 4.6: Inputs and output (Pavlovian conditioning). a,b) CS1 is followed by US; CS2 is not present. c) The output spikes of the neuron when only US is present before learning. d) The output spikes of the neuron without the presence of US after learning.

4.4.2 Extinction

Before the experiment for extinction, Pavlovian learning was first conducted for 10 trials to associate CS1 with US. Then US is removed. During learning, the weight associated with CS1 gradually decreases until it arrives at about 0.15 as figure 4.8 presents. Although the final weight associated with CS1 after learning has not gone down to 0, the extinction of conditioning has been realised because no output spikes are produced by CS1 after learning, viz. the association of CS1 with US has become extinct. When there are no output spikes, there is no learning and therefore the final weight associated with CS1 has not gone down to 0.

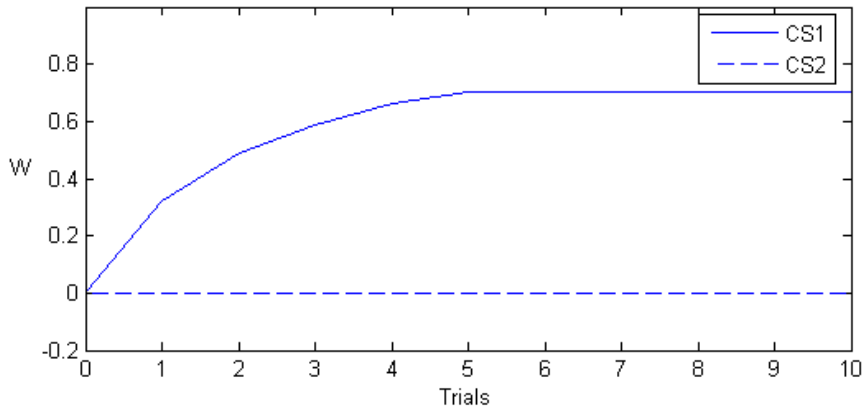

Figure 4.7: Weight updates during learning (Pavlovian conditioning). The weight associated with CS1 keeps increasing until it reaches a saturation value near 0.7 and the weight associated with CS2 remains unchanged. CS1 has been successfully associated with US after learning.

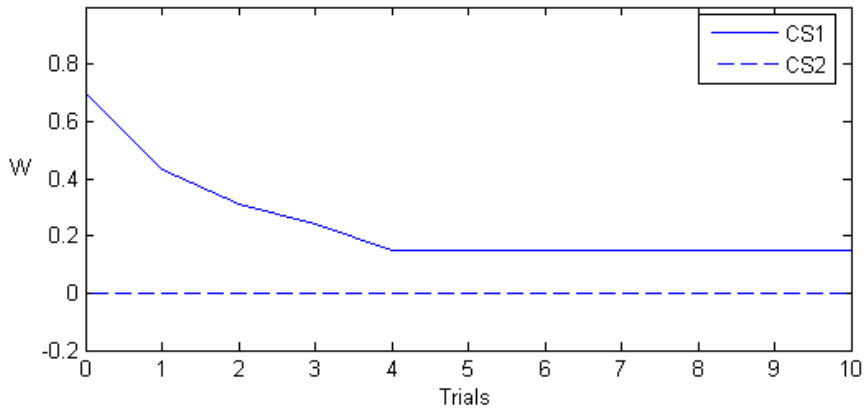

Figure 4.8: Weight updates during learning (extinction). The weight associated with CS1 gradually decreases until it arrives at about 0.15. The association of CS1 with US has become extinct after learning.

4.4.3 Blocking

Similar to the experiment for extinction, CS1 was first associated with US by conducting Pavlovian learning for 10 trials before the simulation of blocking. Then CS2, which has the exact same time trajectory with CS1, is presented as figure 4.9a, b & c depict. During learning, the weight associated with CS1 goes slightly down and the weight associated with CS2 goes slightly up at the beginning but is stabilised at around 0.2 after 1 trial as shown in figure 4.10. In order to see if blocking has been realised, we checked the output of the neuron

with input CS2 only and found that there is no output spike produced throughout the experiment. Therefore, blocking has been realised, even though the weight associated with CS2 is not equal to 0. Comparing figure 4.9d with figure 4.6d, we can see CR produced by both CS1 and CS2 in blocking is almost the same with that produced by CS1 alone in Pavlovian conditioning after learning, which further demonstrates that CS2 has been blocked.

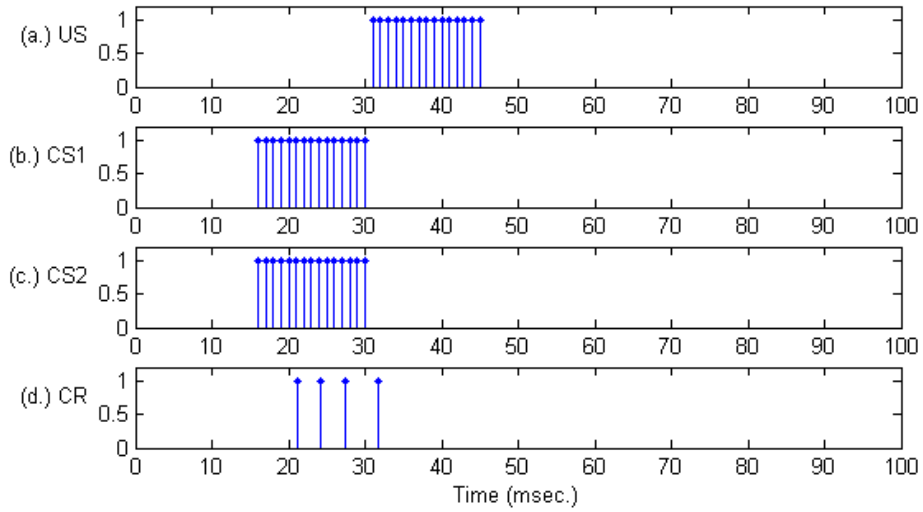

Figure 4.9: Inputs and output (blocking). a-c) The two CS, which have the exact same time trajectory, are followed by US. d) The output spikes of the neuron without the presence of US after learning.

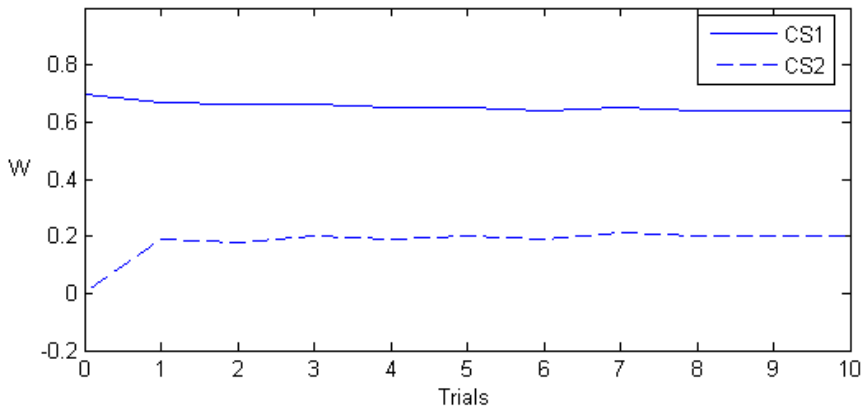

Figure 4.10: Weight updates during learning (blocking). The weights associated with CS1 and CS2 slightly change, but blocking has been realised because CS2 cannot produce any spikes with a maximum weight about 0.2.

4.4.4 Secondary conditioning

Similar to the experiments for blocking and extinction, Pavlovian conditioning was first used for 10 trials to associate CS1 with US. Then CS2, which precedes CS1, is presented and at the same time US is removed as shown in figure 4.11. From the learning curve depicted in figure 4.12, it can be seen that the weight associated with CS1 gradually reduces to 0, and the weight associated with CS2 goes up at first but eventually goes down to a very small amount (about 0.2). After learning, there are no output spikes, which means the associations of both CS1 and CS2 with US have become extinct.

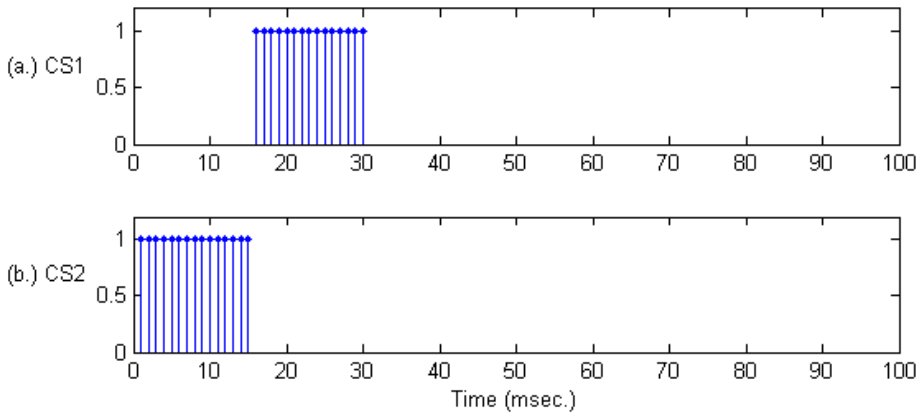

Figure 4.11: Inputs and output (secondary conditioning). CS1 is followed by CS2; US is not present. There are no output spikes after learning.

4.5 Conclusions and discussion

In this chapter, we have used a very simple spiking neuron model to successfully implement the four most important phenomena associated with classical conditioning. Because of the simplicity and effectiveness of the spiking neuron model, it seems that spiking neuron models are well suited to implement classical conditioning and the dynamics of biological neurons offers more power to model the dynamic nature of classical conditioning.

In this section, we will discuss the robustness of the model, the comparison of the learning algorithm used in our model with TD learning, the novelty of our model, an alternative model, and a possible implementation of instrumental conditioning and general reinforcement learning.

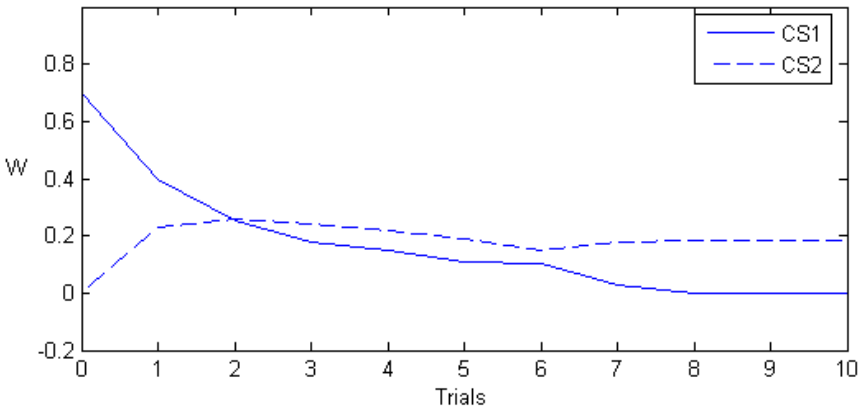

Figure 4.12: Weight updates during learning (secondary conditioning). The weight associated with CS1 gradually reduces until to 0, and the weight associated with CS2 goes up at the beginning but eventually goes down to a very small amount (about 0.2): the associations of both CS1 and CS2 with US have become extinct after learning.

4.5.1 Robustness of the model

Although only one set of experimental scenarios is presented in this chapter, the results still hold with different settings of inputs and different initial weights.

In the simulation of Pavlovian conditioning, for instance, all spikes of CS1 precede all spikes of US and the first spike of US is just after the last spike of CS1. In fact, spikes of CS1 and US may be overlapped or uncontigeous. As long as some spikes of CS1 precede spikes of US within the learning window, the result does not change, though the weight associated with CS1 may be more or less as figure 4.13 and 4.14 show for overlapped inputs and as figure 4.15 and 4.16 show for uncontigeous inputs.

As far as the initial weights associated with conditional stimuli are concerned, they are set to 0 in the simulation. However, they can be very small positive values or very big positive values (even more than 1). The values of initial weights only affect the speed of learning but have no influence on the result.

4.5.2 Compared with TD learning

The learning algorithm 4.13 used in our model is quite like TD learning. In fact, we can consider $\sum_{j=1\cap j\neq i}^{N} \alpha \int_t^{t+T} dt' \int_t^{t+T} dt'' J_j(t) W^{in}(t''-t') S_i^{in}(t'') S_j^{in}(t')$ in our model to be equivalent to $\alpha re(u)$ in TD Learning; $\sum_{j=1\cap j\neq i}^{N} \alpha \int_t^{t+T} dt' \int_t^{t+T} dt''$

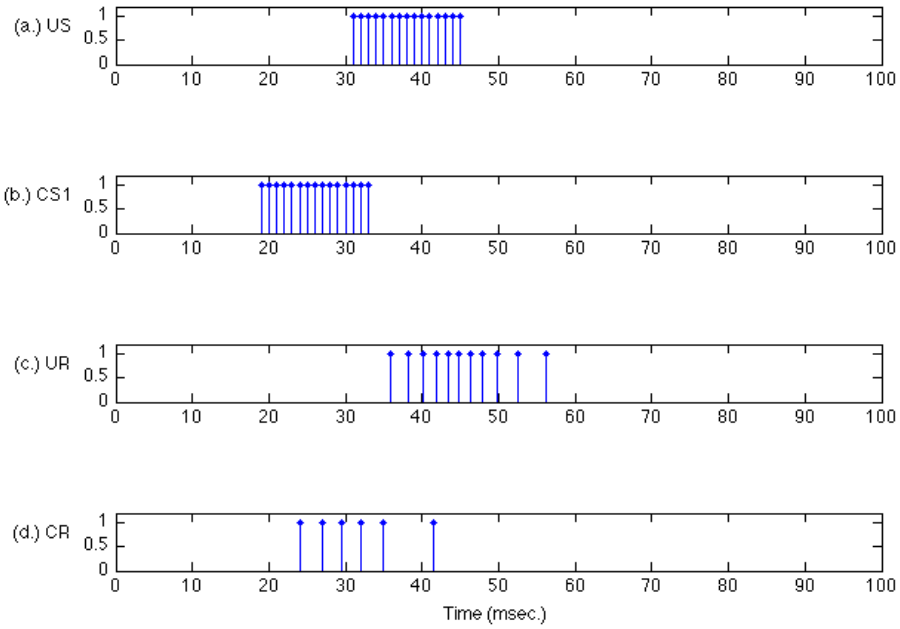

Figure 4.13: Inputs and output (Pavlovian conditioning, overlapped inputs). a,b) CS1 is followed by US; there are overlaps between CS1 and US; CS2 is not present. c) The output spikes of the neuron when only US is present before learning. d) The output spikes of the neuron without the presence of US after learning.

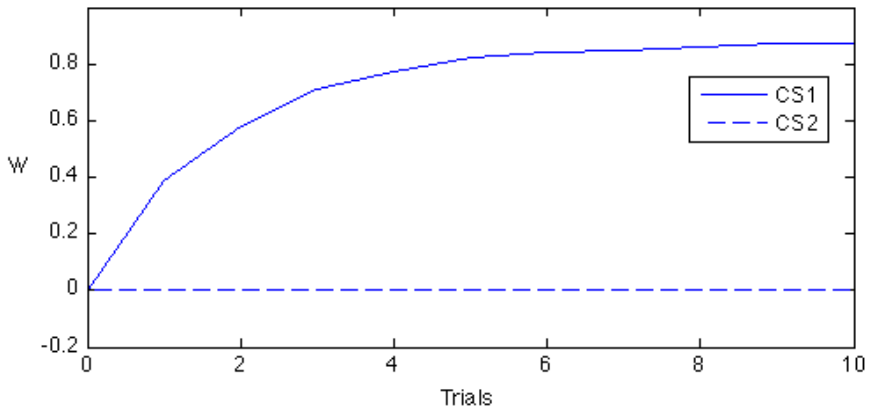

Figure 4.14: Weight updates during learning (Pavlovian conditioning, overlapped inputs). The weight associated with CS1 keeps increasing until it reaches a saturation value near 0.8 and the weight associated with CS2 remains unchanged. CS1 has been successfully associated with US after learning.

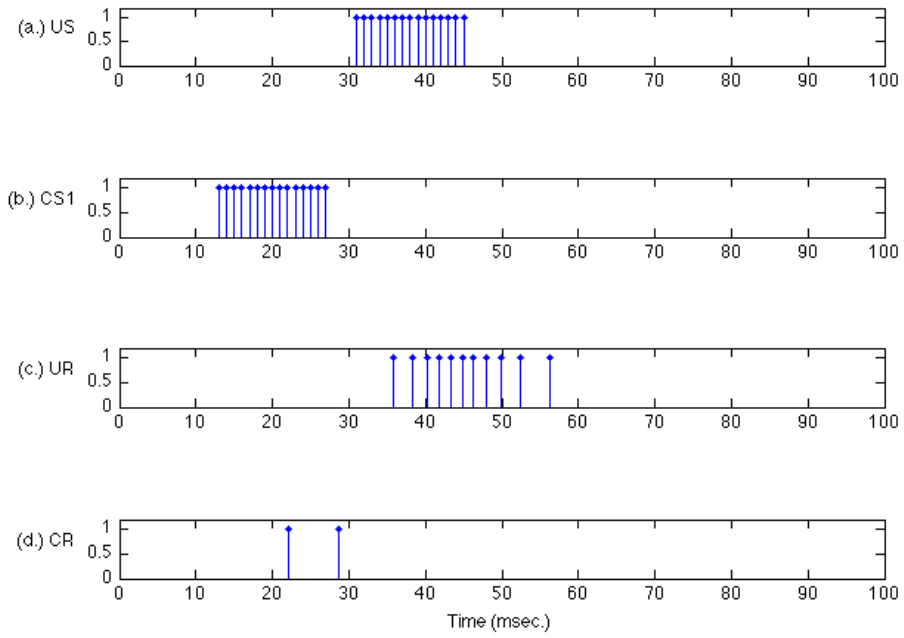

Figure 4.15: Inputs and output (Pavlovian conditioning, uncontigeous inputs). a,b) CS1 is followed by US; there are gaps between CS1 and US; CS2 is not present. c) The output spikes of the neuron when only US is present before learning. d) The output spikes of the neuron without the presence of US after learning.

$W^{out}(t''-t')S_i^{in}(t'')S^{out}(t')$ in our model equivalent to $\alpha[\gamma V(s')-V(s)]e(u)$ in TD Learning.

However, our model is arguably simpler than TD learning. It uses simple spike-time dependent Hebbian learning and spike-time dependent anti-Hebbian learning which are commonly found in biological neurons. In addition, the eligibility trace, which is essential in the TD model, is not needed in our model.

4.5.3 Novelty of the model

As far as we know, we were the first to use spiking neuron models, a kind of biologically more plausible neuron model, to implement classical conditioning.

In addition, as far as we are aware, we were also the first to use Hebbian and anti-Hebbian learning together. Hebbian learning has been used in linear

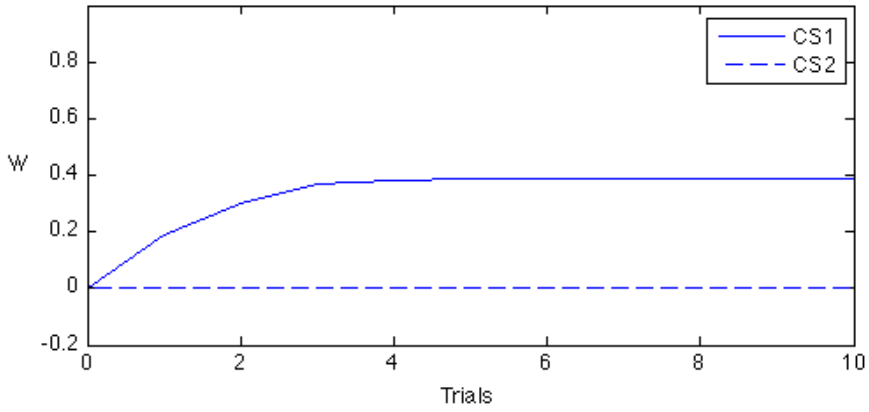

Figure 4.16: Weight updates during learning (Pavlovian conditioning, uncontigeous inputs). The weight associated with CS1 keeps increasing until it reaches a saturation value near 0.4 and the weight associated with CS2 remains unchanged. CS1 has been successfully associated with US after learning.

associator [115], associative learning [116], competitive networks [117] and Grossberg networks [118] in the context of artificial neural networks. In the context of spiking neuron models, Hebbian learning has been used to implement BCM learning rule [155], to learn short synfire chains [156], to detect the coherence in the input [157], to cluster data [158–160], to identify diabetic objects, to model the somatosensory system that has the ability to self-organize [161], to control a modular robotic system [162], and to train competitive networks [163]. Furthermore, Carnell [164] applied both Hebbian and anti-Hebbian learning separately to recurrent spiking neural networks and found that Hebbian and anti-Hebbian learning can be considered approximately equivalent under specific conditions. None of the previous work, however, has used Hebbian and anti-Hebbian learning together as far as we are aware.

Not only is our use of Hebbian and anti-Hebbian learning together innovative, it is also necessary to reproduce the phenomena associated with classical conditioning that we display with our model. If only Hebbian learning is used, the weight of the conditional stimulus will keep increasing without limit in Pavlovian conditioning as figure 4.17 shows and moreover extinction cannot be produced as figure 4.18 shows. On the other hand, if only anti-Hebbian learning is used, the weight of the conditional stimulus will not increase in Pavlovian conditioning and therefore Pavlovian conditioning cannot be formed as figure 4.19 shows.

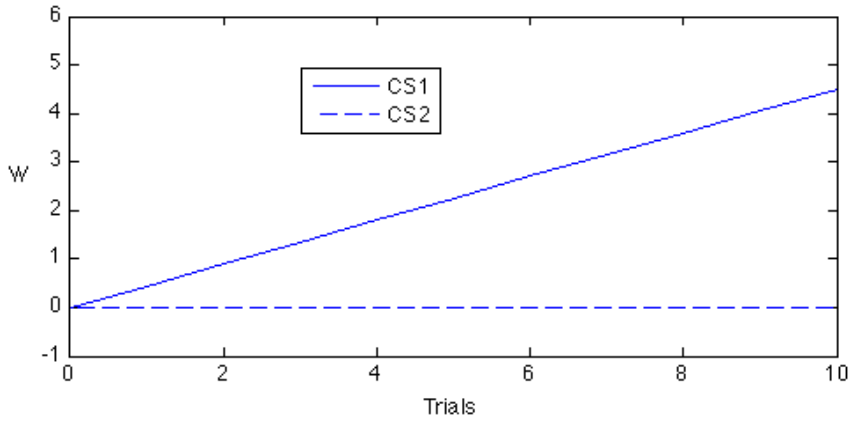

Figure 4.17: Weight updates during learning (Pavlovian conditioning, Hebbian learning only). The weight associated with CS1 keeps increasing without limit.

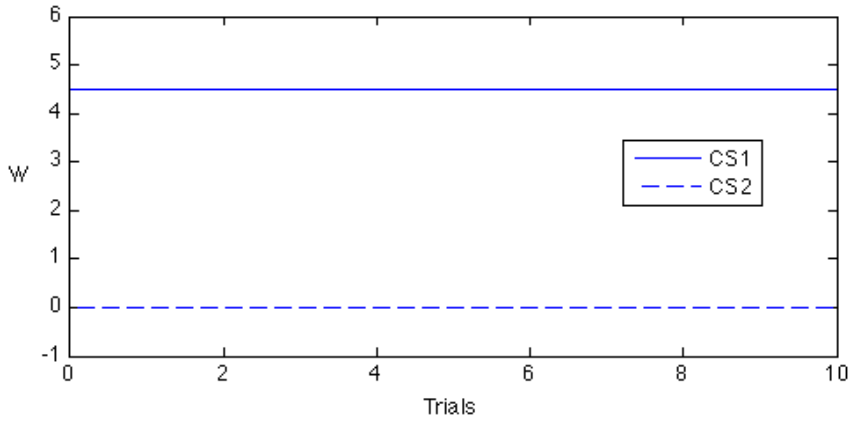

Figure 4.18: Weight updates during learning (extinction, Hebbian learning only). The weight associated with CS1 does not decrease at all.

4.5.4 An alternative model

Instead of requiring learning both between inputs and between the inputs and the output, we can also implement classical conditioning by using only learning between the inputs and the output. Spike-time dependent Hebbian learning is applied between inputs and the output spikes produced by other inputs, and spike-time dependent anti-Hebbian learning is applied between inputs and the output spikes produced by themselves. In addition, as mentioned previously, in biological experiments, it is also the case that spike-time dependent Hebbian learning is formed between the presynaptic spikes and the postsynaptic action potentials produced by other inputs (the current injection) rather than the action

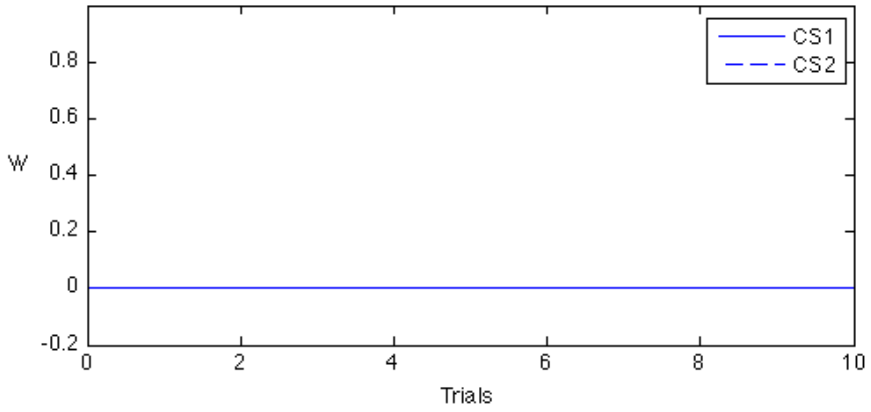

Figure 4.19: Weight updates during learning (Pavlovian conditioning, anti-Hebbian learning only). The weight associated with CS1 does not increase at all.

potentials produced by the presynaptic spikes themselves.

Despite these advantages, however, it is hard to tell which output spike is contributed by which input in practice, since one output spike may be contributed to by more than one input.

4.5.5 Instrumental conditioning and general reinforcement learning

As mentioned previously, in addition to classical conditioning simulated in this chapter, there is also another kind of conditioning, viz. instrumental conditioning. Unlike the classical conditioning we have discussed, in the process of instrumental conditioning, the actions of animals can also have an influence on how many rewards they can obtain.

Our model can be used to implement instrumental conditioning as well without any modification. Suppose that there are n buttons/actions. After pressing one button, an animal will be given a reward. After many repetitions of the experiment, the animal will be able to find the button which gives it the maximum rewards and therefore presses it most of the time (a non-greedy policy) or exclusively (a greedy policy) in order to get the maximum payoff. We can use n independent single neurons used in our model to simulate this conditioning as figure 4.20 shows (only two neurons are drawn for simplicity). Every neuron represents one action/button. The input is a special event/stimulus that signals

the start of a trial and the reward is a kind of special input just like US. The output is the response. After many repetitions of conditioning on these neurons, the action corresponding to the neuron that fires first with the same input will be chosen most of the time (a non-greedy policy) or exclusively (a greedy policy).

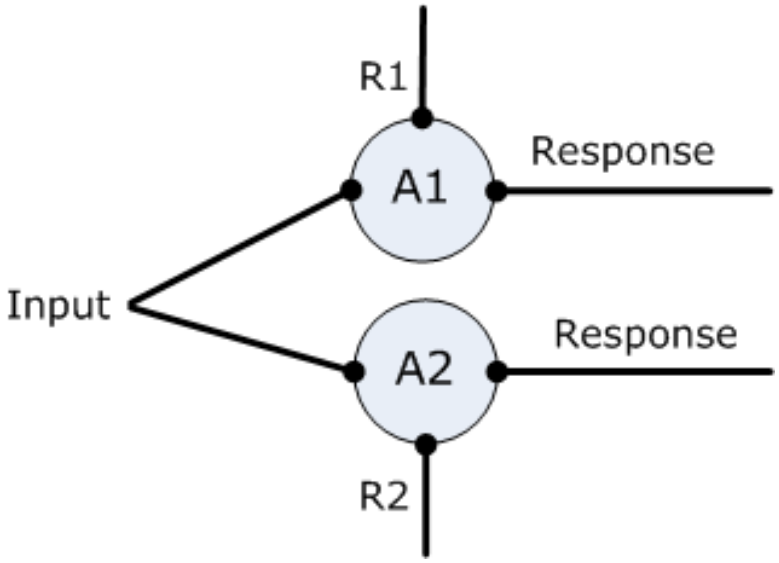

Figure 4.20: A neuron structure modelling instrumental conditioning. Input: a special event/stimulus that signals the start of a trial; A_n: possible actions modelled by neurons; R_n: a reward received when the nth action is chosen; Response: the response to the input or reward.

After implementing instrumental conditioning, it is not difficult to simulate general reinforcement learning using spiking neurons. We use the reinforcement learning scenarios shown in figure 4.21 as an example. The learning agent is initially in state s_0. After taking action a_1, it will receive a reward r_1 and enter state s_1 and so on. On the other hand, if it takes action a_2, it will receive reward r_2 and enter state s_2. We can just chain the neuron structure to model one-step reinforcement learning (only the reward received immediately is counted) as figure 4.22 (a.) shows. In order to model a non-bootstrap (Monte Carlo style) n-step reinforcement learning or even infinite-horizon discounted reinforcement learning, we need to inject rewards received later into the previous states as shown in figure 4.22 (b.). A decreasing learning window as shown in figure 4.23 can serve as a discounted function naturally. It is interesting to point out that the Hebbian learning part of the learning window is also one curve of spike-time dependent Hebbian learning that is found in biological experiments [127]. In order to model a bootstrap reinforcement learning algorithm, e.g. Q learning and

SARSA learning, we need to feed the output of the neuron representing the next state back to the neuron representing the previous state and also feed the output of each neuron to itself (self-feedback), viz. a recurrent network, so that the value of the next state may influence that of the previous state and the previous value of one state may also influence the current value of the state.

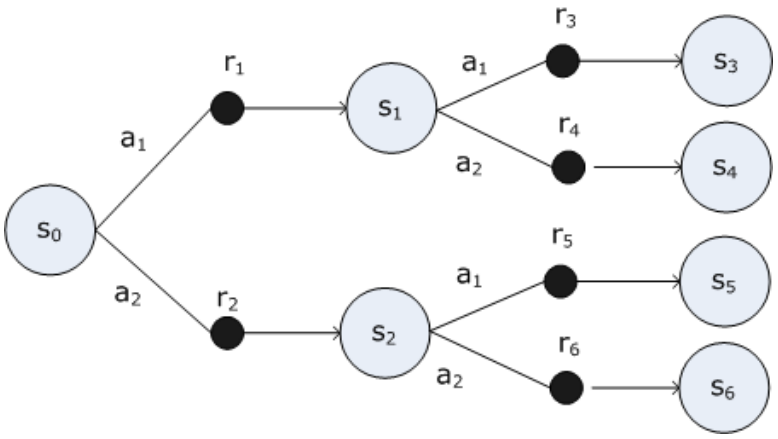

Figure 4.21: A scenario of general reinforcement learning. s: states; a: actions in a state; r: rewards received after taking one action.

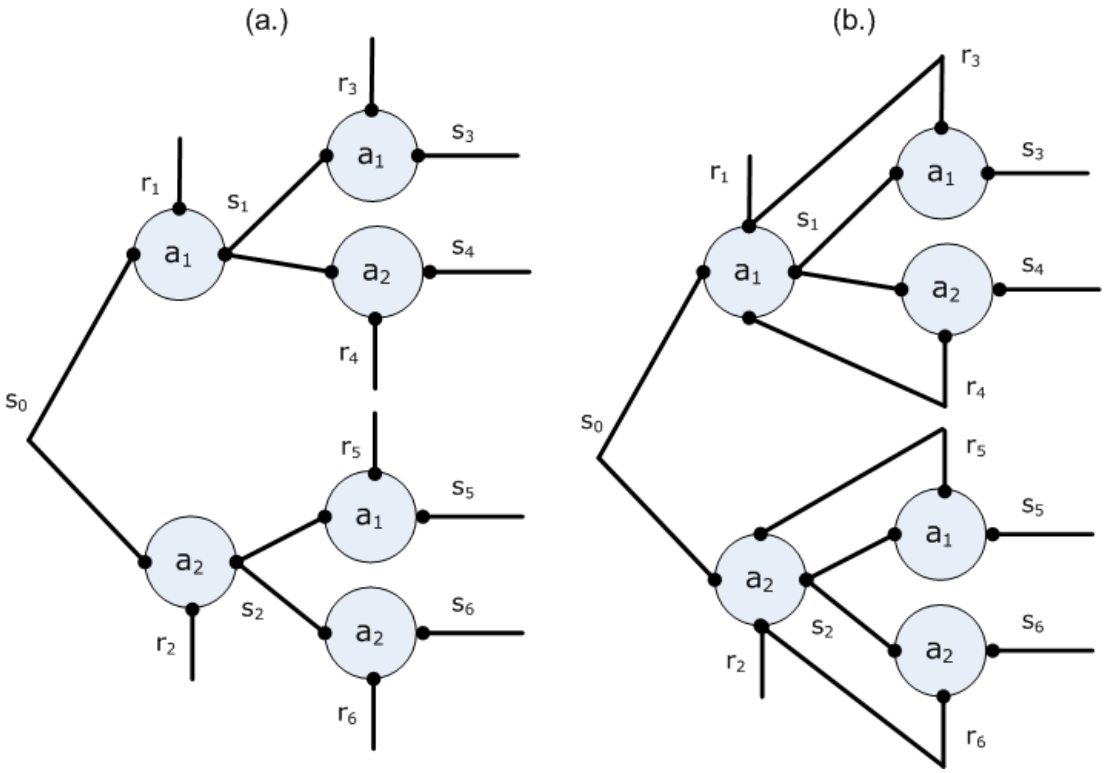

Figure 4.22: The neural structure modelling the general reinforcement learning scenario shown in figure 4.21. (a.) the neural structure modelling one-step reinforcement learning; (b.) the neural structure modelling n-step or even infinite-horizon discounted reinforcement learning. s: states, inputs of neurons; a: actions in a state, modelled by a neuron; r: rewards received after taking one action.

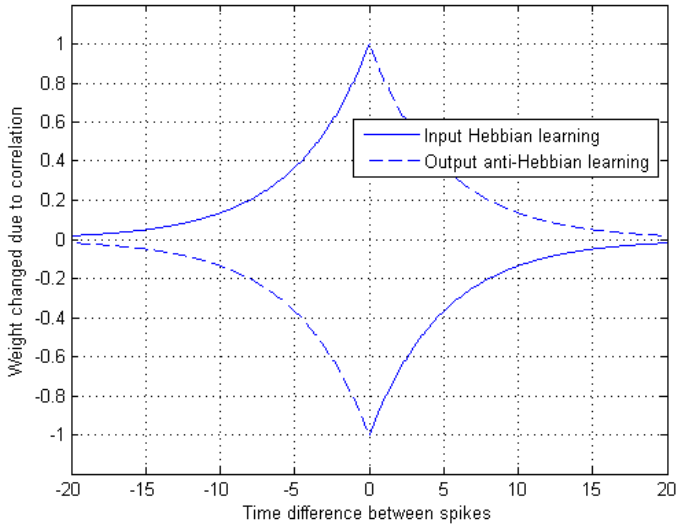

Figure 4.23: A learning window that can serve as a discount function.

Chapter 5

Time delayed n-armed bandit problem

From this chapter on, instead of implementing the idea of time perception with low-level biological models first and then testing whether it works or not, we start by studying the details of time perception and then test its effectiveness by using abstract reinforcement learning models and a perfect clock. The first reason for this decision is that, if we implement it with low-level biological models, the result is implementation dependent. If it does not work, it is hard to determine whether the problem is because of the biological implementation or because of the principle (the idea itself). Besides, for the idea itself, there are still many questions to answer, many problems to address, and many details to investigate. These questions are much easier to address with abstract models rather than low-level biological models.

Regarding detecting any changes in the environment, what should be learned in order to detect changes in the environment? Is learning only the mean of the time to reward sufficient? When the amount of reward may also change, it is obvious that the learned time information cannot detect changes in the amount. In this case, what should the learning agent learn to detect both changes in the time to reward and changes in the amount of reward? On the other hand, even if the amount of reward does not change, learning the mean of the time to reward alone cannot detect changes in the time to reward immediately (in one trial) when the environment is stochastic. In this case, are there any ways to detect changes in the time to reward immediately? Furthermore, after a change in the environment is detected, what should the learning agent do in order to recover

from the change quickly?

Regarding giving up a suboptimal action to avoid longer than necessary exploration, how can the agent decide whether or not and when to give up the current action? If it decides to give up the action, what state should it give up to? After giving up, what action should the agent choose? These questions will be addressed in this and the following chapters.

In the following sections, the time delayed n-armed bandit problem is first introduced. Then, the above questions are answered and possible implementations of the ideas of time perception are investigated and simple algorithms specifically for this kind of reinforcement learning problem with only one state are designed. From section 5.3 to section 5.5, the settings of experiments are discussed and the results of experiments on these algorithms are presented. The last section concludes this chapter and discusses related questions.

5.1 Introduction

The classical n-armed bandit problem assumes that a reward is paid off immediately after an arm is pushed. This assumption, however, is not the case in some real-world situations. For instance, patients in a clinical trial may not be able to give immediate responses after being treated [165] and a supercomputer needs some time to process a task after the task is chosen [166]. In this chapter, we study a more general problem, the n-armed bandit problem with time delay, viz. the time delayed n-armed bandit problem.

Instead of receiving rewards immediately, the learning agent needs to wait some time after pushing an arm and before a reward is received. Both the amount of reward and the time to reward may be deterministic or stochastic and their distributions may also change over time. Though this problem can still be considered as a MDP by augmenting the state space with time steps, we model it as a semi-MDP in order to simplify it from a multiple-state problem to a single-state problem and therefore speed up learning and planning. This problem, however, is only a simplified version of semi-MDP: a reward only appears at a specific and discrete time, viz. at the end of one episode. In addition, instead of considering the temporally extended actions or state transitions as indivisible units, which is the case for the theory of semi-MDPs, we treat them as temporal abstractions of

an underlying MDP so that we can interrupt and change the course of the temporally extended actions or state transitions, similar to Sutton et al.'s framework of options [72, 73]. After choosing one action, the agent has the option to give up the action, go back and remake its choice at every time step while waiting for a reward.

The time delayed n-armed bandit problem, which has only one state but multiple actions, is probably the simplest reinforcement learning problem with delayed rewards. In this chapter, we will experiment with this simple problem and see if learning and perceiving the time to reward can improve the performance of the learning agent in various experimental scenarios.

The time delayed n-armed bandit problem can be described as follows. There is an n-armed bandit and the learning agent can push any of its arms. After one arm is pushed, a reward (r) comes after a period of time (t). Here, we only consider it as a one-step episodic semi-Markov task for simplicity. One episode starts when one arm is pushed and ends when a reward is received. After choosing one action, the agent has the option to give up the action, go back and remake its choice at every time step while waiting for a reward. After receiving a reward, the learning agent goes back and starts another episode. The goal may be to maximise the expected one-step undiscounted return

$$E(r), \tag{5.1}$$

to minimise the expected time to reward

$$E(t), \tag{5.2}$$

to maximise the expected one-step discounted return

$$E(\gamma^t r), \tag{5.3}$$

where γ is the discount rate, $0 \leq \gamma \leq 1$, or to maximise the expected rate of return

$$E(r/t). \tag{5.4}$$

This problem is similar to the optimal foraging problem [20,21]. In the optimal foraging problem, there are n patches and a predator chooses one of the patches to exploit. The possible goals are similar to those of the time delayed n-armed

bandit problem, e.g. to get the maximum rate of reward. In fact, the optimal foraging problem can be simplified to the time delayed n-armed bandit problem under the following three assumptions.

1. Assume that the time needed to reach a patch and switch between patches is very small compared with the waiting time in any patch and therefore it can be ignored for the simplicity of analysis.

2. Assume that there is no energy loss for waiting in one patch and switching between patches apart from the time loss.

3. Assume that the time starts with the selection of a patch. For example, suppose there are two patches, one with the time to reward 3 mins, the other one with the time to reward 4 mins. If the predator waits 3 mins at the first patch and then switches to the second one, it still needs to wait for 4 mins before a reward appears (not 1 min).

5.2 Algorithms

In this section, we introduce simple versions of algorithms specifically designed to solve reinforcement learning problems with only one state, to which the time delayed n-armed bandit problem belongs. In the next chapter, we extend these algorithms to work with multiple states.

We first consider cases where the amount of the reward for actions is the same and also does not change (stationary and deterministic). In these cases, the time to reward for actions can be learned and then used to make decisions. Instead of choosing the action with the maximum value of the estimated discounted reward most of time, the learning agent selects the action with the minimum value of the estimated time to reward most of time. It is worth pointing out that, although it may not be equivalent to maximising the expected discounted reward in stochastic environments, minimising the expected time to reward (getting a reward in the shortest time) is also a sensible optimal target. We introduce three algorithms for these cases, viz. time estimation (T learning), time estimation with time perception (TP learning) and time estimation with time perception without giving up (TPWG learning). T learning is a standard value estimation reinforcement learning algorithm that learns the time to reward (T). In particular, it only learns the mean of the time to reward and then uses it to make decisions. TP/TPWG

learning, however, learns both the mean and variance of the time to reward. In addition to using the estimated mean of the time to reward to make decisions like T learning, they also use both the estimated mean and variance of the time to reward to detect any changes in the environment. When a change is detected, the learning agent responds to it specifically in order to recover from it quickly. In addition, unlike T/TPWG learning, TP learning also uses the estimated mean and variance of the time to reward to find out when the agent should give up this time's exploration of the current action in order to avoid longer than necessary exploration.

Next, we consider cases where the amount of reward for actions may be different and may also change over time. In these cases, learning the time to reward alone is neither enough to make decisions nor enough to detect changes in the environment. Thus, the discounted reward is learned and then used to make decisions instead. In addition, we extend the ideas of learning and perceiving the time to reward to learning and perceiving the discounted reward. We introduce three algorithms for these cases, viz. value estimation (V learning), value estimation with value perception (VP learning) and value estimation with value perception without giving up (VPWG learning). V learning is a standard value estimation reinforcement learning algorithm that learns the discounted return. In particular, it only learns the mean of the discounted reward and then uses it to make decisions. VP/VPWG learning, however, learns both the mean and variance of the discounted reward. In addition, VP learning also learns both the mean and variance of the undiscounted reward. In addition to using the estimated mean of the discounted reward to make decisions like V learning, VP/VPWG learning also uses both the mean and variance of the discounted reward to detect changes in the environment. In addition, VP learning uses all learned information to decide whether to give up this time's exploration of the current action.

It is worth noting that both T learning and V learning are not our original contributions but standard value estimation reinforcement learning algorithms used to compare with our learning algorithms.

The notation used to describe algorithms is summarised in table 5.1.

Table 5.1: Summary of notation used to describe algorithms for the time delayed n-armed bandit problem

Notation	Meaning
A	set of possible actions
a	any action $\in A$
$A \backslash a$	set of possible actions except action a
a_c	the current action
a^*	the optimal action in terms of the agent's criterion of optimality
a_g	the action the agent will choose after giving up; 0 (viz. no action) if it should not give up
α	learning rate, $0 < \alpha \leq 1$; a scalar with a fixed value for T and V learning and a function of actions for both TP/TPWG and VP/VPWG learning
α_0, δ, η	parameters used to calculate α for TP/TPWG and VP/VPWG learning
$\alpha_2(a)$	learning rate specifically used to learn the variance for action a, $0 < \alpha_2 \leq 1$
α_{2max}, α_{2min}	the maximum/minimum of α_2
ϵ	random parameter to explore suboptimal actions, $0 \leq \epsilon \leq 1$
ϵ_{max}, ϵ_{min}	the maximum/minimum of ϵ
ϕ	parameter used to calculate ϵ in both TP/TPWG and VP/VPWG learning
ϵ_2	random parameter to decide whether to explore the current action longer than usual; with probability $\frac{\epsilon_2}{2}$, explore the current action longer than usual, $0 \leq \epsilon_2 \leq 1$

ϵ_3	random parameter to decide whether to explore the amount of reward for the current action; with probability $\frac{\epsilon_3}{2}$, the agent does not give up the current action, $0 \leq \epsilon_3 \leq 1$
γ	discount rate to discount future rewards, $0 \leq \gamma \leq 1$
t	discrete time step
r	the amount of reward
$T(a)$	estimated mean of the time to reward for action a
$T\_var(a)$	estimated variance of the time to reward for action a
$Q(a)$	estimated mean of the discounted reward for action a
$Q\_var(a)$	estimated variance of the discounted reward for action a
$R(a)$	estimated mean of the undiscounted reward for action a
$R\_var(a)$	estimated variance of the undiscounted reward for action a
k	the size of the expectation window ranging from $T(a) - k\sqrt{T\_var(a)}$ to $T(a) + k\sqrt{T\_var(a)}$, $Q(a) - k\sqrt{Q\_var(a)}$ to $Q(a) + k\sqrt{Q\_var(a)}$, or $R(a) - k\sqrt{R\_var(a)}$ to $R(a) + k\sqrt{R\_var(a)}$
$count\_correct(a)$	the number of times/episodes in a row that the actual time to reward or the actual discounted reward for a is correctly estimated; whether they are correctly estimated is judged by algorithms.
$threshold(a)$	the time step after which the learning agent should give up action a
$action\_forCmp(a)$	the action which the agent uses to compare with action a; 0 (viz. no action) if no action is eligible
$flag\_correctEst(a)$	a Boolean storing the information about whether or not the actual time to reward or the actual discounted reward for a is correctly estimated judged by algorithms; refer to section 5.2.2.5 for how to evaluate whether or not they are correctly estimated

$flag\_correctEstR(a)$	a Boolean storing the information about whether or not the estimated amount of reward for action a is correct judged by algorithms
$flag\_correctCmp(a)$	a Boolean storing the information about whether or not action a can be used to compare with the current action judged by algorithms
$flag\_exploreAmount$	a Boolean deciding whether the learning agent will explore the amount of reward, viz. not give up

5.2.1 Time estimation

The mean of the time to reward is learned through an incremental updating rule with a fixed learning rate. The mean of a random variable x can be estimated from a finite sample of observations incrementally using

$$m_n = \frac{x_n + m_{n-1}(n-1)}{n} = m_{n-1} + \frac{x_n - m_{n-1}}{n} \tag{5.5}$$

where m is the estimated mean, x_n is the nth observed data, m_n is the nth estimation of the mean. In order to avoid tracking/memorising n (the number of data observed so far) and also to track nonstationary environments as discussed in subsection 3.5.1, we replace $\frac{1}{n}$ with a very small number α:

$$m_n \approx m_{n-1} + \alpha(x_n - m_{n-1}) \tag{5.6}$$

where $0 < \alpha \leq 1$ is the learning rate for learning the mean.

As shown in algorithm 1, a standard time estimation reinforcement learning algorithm (T learning) learns the time to reward (T) through an incremental updating rule with a fixed learning rate shown in equation 5.6 and then uses it to make decisions through a non-greedy rule, e.g. ϵ-greedy and softmax methods. Since the n-armed bandit problem has only one state (but n actions) and we only consider it as a one-step episodic task, it has no state transition and the T value is only updated when a reward is received. In essence, this algorithm is just a simplified version of classical reinforcement learning algorithms to learn the time to reward with only one state.

[1]The action with the minimum T is chosen most of time, but with probability ϵ, actions are chosen randomly.

Algorithm 1 T learning

Inputs: T; **Outputs:** T; **Parameters:** α, ϵ; **Internal variables:** t
for all episodes **do**
  Choose a_c from all possible actions using the policy derived from T (ϵ-greedy, minimum T priority[1])
  $t = 0$
  repeat
    $t \leftarrow t + 1$
  until a reward is received
  $T(a_c) \leftarrow T(a_c) + \alpha\,[t - T(a_c)]$
end for

5.2.2 Time estimation with time perception

In addition to learning the mean of the time to reward as in the standard time estimation reinforcement learning algorithm discussed in the last subsection, the algorithms of time estimation with time perception (TP learning) and time estimation with time perception without giving up (TPWG learning) also learn the variance of the time to reward as shown in algorithm 2-4. The learned time information including both the estimated mean and variance of the time to reward is then used to detect changes in the environment. If a change in the environment is detected, the learning rate is increased in order to learn the change quickly. Otherwise, the learning rate is decreased gradually towards 0 in order for the estimated mean of the time to reward to converge to its true mean. If a suboptimal action has improved and can potentially become the optimal action in the new environment, the exploration rate (e.g. ϵ for ϵ-greedy) is also increased in order to increase the chance that the suboptimal action is visited. Otherwise, the exploration rate is decreased gradually towards its minimum value in order to reduce the cost of exploring suboptimal actions. In addition, for TP learning only, if the agent discovers that the current action is still worse than the optimal one, it gives up this time's exploration of the action in order to avoid longer than necessary exploration. Therefore, TPWG learning is exactly the same with TP learning except that the learning agent does not give up. This algorithm is designed to make the experimental comparison fairer because giving up or not giving up is to be regarded as part of the problem formulation, not as part of an algorithm. The details of the algorithms will be discussed in the following subsections.

Algorithm 2 TP/TPWG learning

Inputs: T, T_var; **Outputs:** T, T_var;
Parameters: k; **Internal variables:** A, a_c, t, ϵ (initial value: ϵ_{min}), $flag\_correctEst$ (initial value: $FALSE$), $count\_correct$ (initial value: 0), a_g (for TP learning only), $threshold$ (initial value: ∞, for TP learning only), $action\_forCmp$ (initial value: 0, for TP learning only), $flag\_correctCmp$ (initial value: $FALSE$, for TP learning only)
for all episodes **do**
  Choose a_c from all possible actions using the policy derived from T (ϵ-greedy, minimum T priority)
  $t = 0$
  $a_g = 0$ {This line is for TP learning only}
  Use algorithm 4 to calculate $threshold(a_c)$ and $action\_forCmp(a_c)$ {This line is for TP learning only}
  repeat
    {Below is for TP learning only}
    if $a_g \neq 0$ **then**
      $a_c = a_g$, $a_g = 0$, $t = 0$
      Use algorithm 4 to calculate $threshold(a_c)$ and $action\_forCmp(a_c)$
    end if
    if $t > T(a_c) + k\sqrt{T\_var(a_c)}$ **then**
      $flag\_correctEst(a_c) = FALSE$
      $flag\_correctCmp(a_c) = FALSE$
    end if
    if $flag\_correctEst(a_c) = TRUE\ AND\ t > threshold(a_c)$ **then**
      $a_g = action\_forCmp(a_c)$ {Give up and then choose a_g}
    end if
    {Above is for TP learning only}
    $t \leftarrow t + 1$
  until a reward is received
  Use algorithm 3 to update the model
end for

Algorithm 3 Update the model (used by algorithm 2)

Inputs: T, T_var, a, ϵ, t, $count\_correct$, $flag\_correctEst$, $flag\_correctCmp$ (for TP learning only)
Outputs: T, T_var, ϵ, $count\_correct$, $flag\_correctEst$, $flag\_correctCmp$ (for TP learning only)
Parameters: α_0, α_{2max}, α_{2min},ϵ_{max}, ϵ_{min}, ϕ, k, δ, η
Internal variables: α, α_2, a^*, T_{old}
$flag\_correctCmp(a) = \begin{cases} TRUE & t \leq T(a) + k\sqrt{T\_var(a)} \\ FALSE & \text{otherwise} \end{cases}$ {This line is for TP learning only}
if $T(a) - k\sqrt{T\_var(a)} \leq t \leq T(a) + k\sqrt{T\_var(a)}$ **then**
  $flag\_correctEst(a) = TRUE$, $\alpha_2(a) = \alpha_{2max}$, $count\_correct(a) \leftarrow count\_correct(a) + 1$
else
  $flag\_correctEst(a) = FALSE$, $\alpha_2(a) = \alpha_{2min}$
  if this happens twice in a row **then**
    $count\_correct(a) = 0$
  end if
end if
$\alpha(a) = \frac{\alpha_0}{(count\_correct(a)+1+\delta)^\eta}$
if a is not the optimal action **then**
  $\epsilon = \begin{cases} \epsilon + \phi\,[\epsilon_{max} - \epsilon] & t < T(a) - k\sqrt{T\_var(a)} \ AND \ t < T(a^*) \\ \epsilon + \phi\,[\epsilon_{min} - \epsilon] & \text{otherwise} \end{cases}$
end if
{Update the estimation}
$T_{old}(a) = T(a)$; $T(a) \leftarrow T(a) + \alpha(a)\,[t - T(a)]$
$T\_var(a) \leftarrow T\_var(a) + \alpha_2(a)\{[t - T_{old}(a)][t - T(a)] - T\_var(a)\}$

Algorithm 4 Calculate when it should give up (used by algorithm 2); for TP learning only

Inputs: T, A, a, $flag\_correctCmp$; **Outputs:** $threshold$, $action\_forCmp$
Parameters: ϵ_2; **Internal variables:** a', A', a''
if $\exists a' \in A \backslash a$ satisfying $flag\_correctCmp(a') = TRUE$ **then**
  use A' to represent the set of all a', $a'' = \arg\min_{a' \in A'} [T(a')]$
  $action\_forCmp(a) = a''$
  if $T(a) \leq 2T(a'')$ **then**
    $threshold(a) = \infty$
  else
    $threshold(a) = \begin{cases} T(a'') & \text{with probability } 1 - \epsilon_2/2 \\ 2T(a'') & \text{with probability } \epsilon_2/2 \end{cases}$
  end if
else
  $action\_forCmp(a) = 0, threshold(a) = \infty$
end if

5.2.2.1 Learn the variance of the time to reward

For the same reasons in the estimation of the mean, we replace $\frac{1}{n-1}$ in equation 3.15 with a very small number α_2:

$$v_n \approx v_{n-1} + \alpha_2[(x_n - m_{n-1})(x_n - m_n) - v_{n-1}] \tag{5.7}$$

where $0 < \alpha_2 \leq 1$ is the learning rate for learning variance.

About the choice of the parameter α_2, it should be less than α used in equation 5.6 to estimate the mean. This is first because the learning of variance fluctuates more than the learning of mean if the same learning rate is used. We have used equation 5.6 and 5.7 with different learning rates to learn the mean and variance of random variables drawn respectively from Poisson distribution shown in figure 5.1 and from normal distribution shown in figure 5.2. From the two figures, we can see that, whereas the mean can be learned in a stable manner with a learning rate of 0.1, the learning of variance fluctuates much more dramatically with the same rate of learning. When the learning rate is 0.01, the learning of variance becomes more stable and closer to the learning of mean with a learning rate of 0.1. If the learning rate further decreases (e.g. 0.001), the learning of variance becomes quite slow.

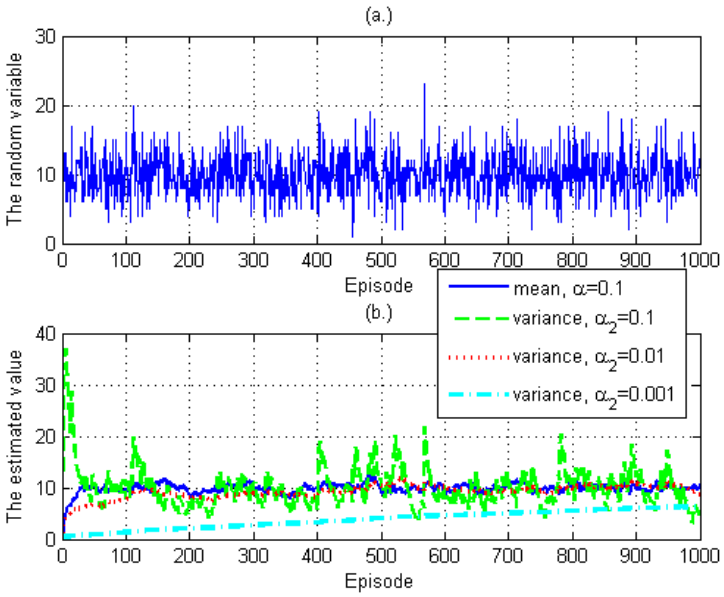

Figure 5.1: Learn the mean and variance of the time to reward (Poisson distribution). (a.) data randomly drawn from Poisson(10); (b.) learn the mean and variance of the data with different learning rates. The learning of variance is much noisier than the learning of mean with the same learning rate.

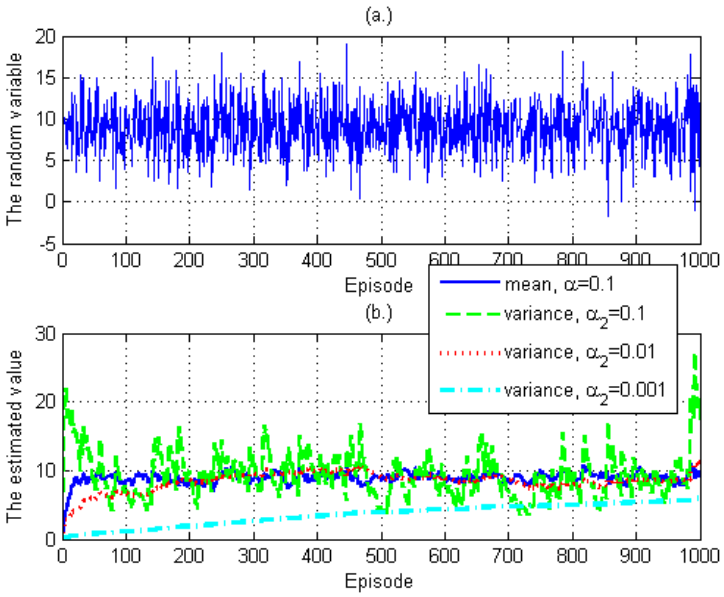

Figure 5.2: Learn the mean and variance of the time to reward (normal distribution). (a.) data randomly drawn from norm(9,3); (b.) learn the mean and variance of the data with different learning rates. The learning of variance is much noisier than the learning of mean with the same learning rate.

5.2.2.2 Detect changes in the environment

When the actual time to reward for arm/action a in the current episode is outside $T(a) \pm k\sqrt{T\_var(a)}$ where $T(a)$ and $T\_var(a)$ are respectively the estimated mean and variance of the time to reward for action a and $k \geq 0$, we consider that the time to reward for action a has changed. When the actual time to reward for action a is less than $T(a) - k\sqrt{T\_var(a)}$, we consider that the time to reward for the action has become shorter. It is worth pointing out that a change in the environment is more subjective to the learning agent than an objective matter. Suppose that the learning agent's initial estimation of the environment is not correct. The learning agent may consider that the environment has changed because the actual environment does not agree with its estimation, even though the environment may never have changed. The agent still needs to reduce the difference between its estimation and the actual value, whether the difference is due to a change of the environment or its incorrect estimation of the environment.

From Chebyshev's inequality [167], we know that no more than $\frac{1}{k^2}$ of the possible values of a random variable are more than k standard deviations away from its mean, no matter what distribution it has. From one-sided Chebyshev's inequality, also known as Cantelli's inequality, no more than $\frac{1}{k^2+1}$ of the values are less than the mean minus k standard deviations. Specifically, if $k = 3$, no more than $\frac{1}{9}$ of the values are more than 3 standard deviations away from the mean, no matter what distribution a random variable has. From one-sided Chebyshev's inequality, no more than $\frac{1}{10}$ of the values are less than the mean minus 3 standard deviations. For a normal distribution, in particular, no more than 3% of the values are more than 3 standard deviations away from the mean. Therefore, we can effectively detect changes in the time to reward in most cases with the above method. Furthermore, we can cover even more cases with a greater standard deviation window. For instance, if we choose a window with 6 standard deviations, no more than 1/36 of the values are outside the window for any distribution and 1/506842372 for normal distribution. This is also the theory behind the famous 6σ business management strategy initially implemented by Motorola [168].

This method has the potential to detect a change in the environment with only one trial even in a stochastic environment. On the other hand, if only the mean of the time to reward is learned, it needs many trials to detect a change in a stochastic environment because it has to compare the mean of the values in recent

several trials with the mean of the values in several trials before recent several trials. Furthermore, this method is much easier to implement than learning the full distribution of the time to reward, which is a non-trivial task in its own right.

However, it is worth noting the following four points about this method. Firstly, no matter how big the window is (except infinity), there is no guarantee that all data from the same distribution are within the window for some distributions, e.g. normal distribution. This means that a false detection of a change is unavoidable. Therefore, the policy used to handle environmental changes needs to accommodate these data, e.g. responding incrementally rather than abruptly. Secondly, when the time to reward changes not very greatly in a stochastic environment, e.g., the new time to reward is still within $T(a) \pm k\sqrt{T\_var(a)}$, this method cannot detect the change. But, if the change in the environment is very small, it may not be necessary to respond specifically to the change, since classical value estimation reinforcement learning algorithms may be able to handle the change very well. In addition, if the mean has not changed but the variance has increased, this method may still consider the environment changed. Lastly, k cannot be too small or too large. A large k can reduce the rate of false detections of environmental changes but at the same time it increases the chance of failing to detect real environmental changes, viz. less sensitive to changes in the environment. On the other hand, a small k can detect even small environmental changes, but at the same time it is more likely to make a false detection of environmental changes, viz. too sensitive to changes in the environment.

5.2.2.3 Respond to a change in the environment

When a change in the environment is detected by the method mentioned above, the agent can increase the learning rate, increase the exploration rate (e.g. ϵ for ϵ-greedy), or give up the current action and then remake decisions in order to adapt to the new environment quickly.

Firstly, the agent can increase the learning rate in order for the agent to learn the change quickly. Whenever a change in the time to reward for one arm a is detected by the above method, the learning rate for this arm is increased either temporarily to a high value or incrementally in order to learn the change quickly. One way to increase the learning rate temporarily is to use a bigger learning rate if a change is detected twice in a row. The reason why we require that a change is detected twice in a row before the learning rate is increased is to

reduce the influence of noise. Alternatively, we can also use a probabilistic rule: the probability of increasing the learning rate is increased with the number of changes detected in a row increasing. Otherwise, a small learning rate is used. One way to increase the learning rate incrementally is, when a change is detected, to update the learning rate by

$$\alpha(a) \leftarrow \alpha(a) + \phi_{\alpha\_in} [\alpha_{max} - \alpha(a)] \tag{5.8}$$

where $\alpha(a)$ is the learning rate of action a, $0 < \phi_{\alpha\_in} \leq 1$, α_{max} is the maximum value of α. Otherwise, it is updated by

$$\alpha(a) \leftarrow \alpha(a) + \phi_{\alpha\_de} [\alpha_{min} - \alpha(a)] \tag{5.9}$$

where $0 < \phi_{\alpha\_de} \leq 1$, α_{min} is the minimum value of α.

Regarding the learning rate (α_2) used to calculate the estimated variance, a smaller value should be used when a change in the environment is detected and a bigger value used otherwise. This is so that the estimated mean converges first and then the estimated variance. Otherwise, the estimated variance becomes so big that the changed actual time to reward may still fall within the estimated mean plus and minus k standard deviations, even though the learning agent has not yet recovered from the environmental change. When the environment changes, two things contribute to the variance, viz., the change in the environment (more precisely, the change in the mean) and the variance of the new environment. In order to reduce the influence of the change in the environment on the estimation of variance, a smaller learning rate is used to learn the variance when the environment changes. Admittedly, it would be ideal to eliminate the influence of the change in the environment on the estimation of variance completely and only learn the variance of the new environment. Unfortunately, however, it seems impossible because the mean of the new environment is unknown.

Secondly, the agent can increase the exploration rate (e.g. ϵ for ϵ-greedy) to increase the chance that suboptimal actions are explored. Suppose that the time to reward for the optimal arm has not changed or has changed to a value still less than the previous time to reward for all other arms. If the new time to reward for another arm has become less than that for the previous optimal arm, it would take the learning agent a long time to find that the suboptimal arm has improved because the suboptimal action is seldom selected. Even after

the arm has been explored, it still takes a long time to update the value of the previous suboptimal action before it can take the place of the previous optimal action, because suboptimal actions are seldom explored and values are updated incrementally. One way to speed up the speed of updating the value of the suboptimal arm is to increase the random parameter ϵ if ϵ-greedy is used to make a non-greedy decision. Specifically, ϵ is increased whenever the actual time to reward for any suboptimal action in the current episode becomes shorter than its estimated mean minus k standard deviations and shorter than the expected time to reward for the optimal action. When these two conditions are satisfied, increase ϵ towards its maximum value by

$$\epsilon \leftarrow \epsilon + \phi_{\epsilon\_in} [\epsilon_{max} - \epsilon] \tag{5.10}$$

where ϵ_{max} is the maximum possible value of ϵ and ϕ_{ϵ_in} is a small positive number. Otherwise,

$$\epsilon \leftarrow \epsilon + \phi_{\epsilon\_de} [\epsilon_{min} - \epsilon] \tag{5.11}$$

is used to decrease epsilon towards its minimum value. Here, ϵ_{min} is the minimum possible value of ϵ and ϕ_{ϵ_de} is a small positive number. In particular, when the environment stays the same, epsilon decreases towards its minimum value and therefore ϵ will converge to ϵ_{min} as long as the environment stays the same for long enough after one change whether the changes on the time to reward among actions are correlated or not.

It is worth noting that the two conditions should both be checked. When the new time to reward for a suboptimal action becomes shorter, the new time to reward may still be much worse than that for the optimal action, where we should not increase ϵ. On the other hand, if the new time to reward for a suboptimal action has not become shorter, it is likely that its time to reward has not changed and therefore is still longer than that for the optimal action. The reason why its actual time to reward in some episodes is shorter than the estimated time to reward for the optimal action may be that there are only some samples shorter whereas most are not. If so, the suboptimal action is still suboptimal and therefore we should not increase ϵ either. Otherwise, ϵ would fluctuate. When both conditions are satisfied, it is more likely that the suboptimal action may have become optimal.

In addition, for problems with more than two arms, increasing ϵ would encourage exploration of all suboptimal actions including those whose time to reward has not changed or has even become longer. In order to solve this problem, a softmax method can be used instead of ϵ-greedy. Only the suboptimal actions, whose time to reward has become shorter and also shorter than the estimated time to reward for the optimal action, are explored more often.

Finally, the agent can give up the current action if the time to reward for the current action has been delayed. If the reward may never come or is so delayed that it is not worth waiting, it makes sense for the agent to give up and choose other actions instead rather than to wait there forever. Unfortunately, however, the learning agent would never know whether the reward will appear in the near future or not unless it continues its waiting and the 'future' arrives.

TP/TPWG learning uses the first two methods, viz. increase the learning rate and increase ϵ, to respond quickly to changes in the environment. The learning agent will not give up actions unless it has learned the new environment correctly.

5.2.2.4 Ensure the estimated mean converges to the true mean

In a deterministic stationary environment, the estimated mean calculated by equation 5.6 with a fixed learning rate, will eventually converge to the true mean regardless of the initial estimated value. In a stochastic stationary environment, however, the estimated mean calculated by equation 5.6 with a fixed learning rate cannot converge to the true mean regardless of the initial estimated value. As figure 5.3 shows, although the initial estimated value is equal to the true mean, the estimated mean still fluctuates even with a moderate fixed learning rate 0.1. With a smaller learning rate, it would fluctuate less seriously at the expense of the learning speed, but the fluctuation cannot be completely eliminated unless the learning rate is equal to 0. Generally speaking, the smaller the learning rate is, the slower the learning speed is, unless the initial estimated mean is already equal to or close to the true mean where it does not need learning or a greater learning rate may push the estimated mean further away from its true value. As discussed previously, when $\alpha(n)$, the value of α used to calculate m_n, satisfies equation 2.21, the estimated mean is guaranteed to converge to the true mean.

As mentioned previously, a fixed learning rate does not satisfy equation 2.21. Does the learning rate calculated by equation 5.9 satisfy the two conditions? If $\alpha_{min} > 0$, $\alpha(n)$ will converge to α_{min} eventually with the equation and $\alpha(n)$ is not

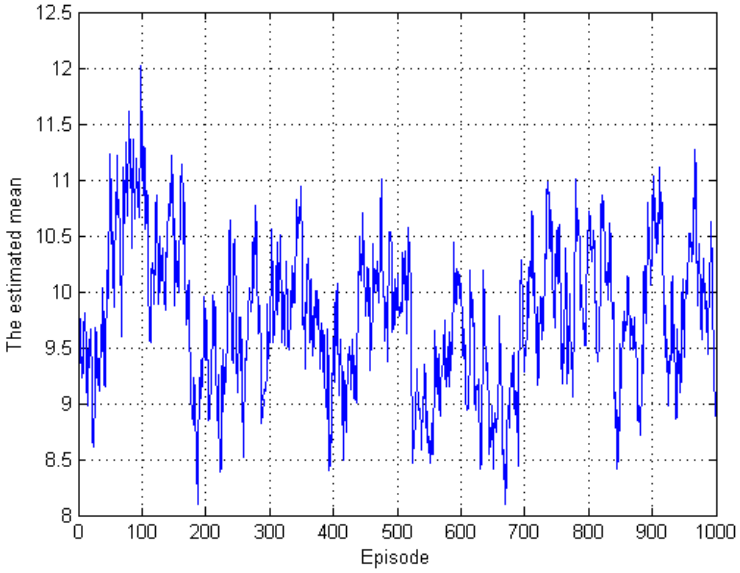

Figure 5.3: Learn the mean of a random variable with a fixed learning rate (0.1); the initial estimation is 10; the data is drawn from a Poisson distribution with a mean of 10. Although the initial estimated value is equal to the true mean, the estimated mean still fluctuates even with a moderate fixed learning rate 0.1.

less than α_{min} for any n assuming that $\alpha_{max} \geq \alpha_{min}$. Based on these observations, we have

$$\sum_{n}^{\infty} \alpha^2(n) \geq \alpha_{min}^2 \sum_{n}^{\infty} n = \infty. \tag{5.12}$$

Therefore, the second condition is not satisfied. On the other hand, if $\alpha_{min} = 0$,

$$\alpha(n) = \alpha(n-1) - \phi\alpha(n-1) = (1-\phi)\alpha(n-1) = (1-\phi)^n\alpha(0). \tag{5.13}$$

Based on this equation, we have

$$\sum_{n}^{\infty} \alpha(n) = \sum_{n}^{\infty} (1-\phi)^n\alpha(0) = \alpha(0) \lim_{n\to\infty} \frac{1-(1-\phi)^n}{\phi} = \frac{\alpha(0)}{\phi} \tag{5.14}$$

if $0 < \phi \leq 1$. Therefore, the first condition is not satisfied. One choice of $\alpha(n)$ which does satisfy both conditions is

$$\alpha(n) = \frac{\alpha(0)}{(n+\delta)^\eta} \tag{5.15}$$

where $\alpha(n)$ is the learning rate used to calculate the nth estimation of mean in equation 5.6, δ is a non-negative number, and $1 \leq \eta < 2$.

On the other hand, the learning rate satisfying equation 2.21 does not work well in nonstationary environments where the learning agent needs to keep learning in case the environment has changed and therefore cannot decrease its learning rate to a very small value. This is also one of the main reasons why a fixed learning rate is usually used in reinforcement learning.

Another disadvantage of using a learning rate satisfying equation 2.21 is that the learning is quite slow even in stationary environments if the initial estimated mean is not close to the true mean and the initial learning rate is not very big. We do experiments on a Poisson distributed random variable with a mean of 10 with different initial estimated mean. $\alpha(n)$ is calculated by equation 5.15 and we set $\alpha(0) = 0.1$, $\delta = 0$ and $\eta = 1$. As figure 5.4 shows, when the initial estimated value is 10, the difference between the estimated mean and the true mean is near 0 after 1000 episodes' learning; when the initial estimated value is 8, the difference between the estimated mean and the true mean is within 1 after 1000 episodes' learning; when the initial estimated value is 12, the difference between the estimated mean and the true mean is more than 1 after 1000 episodes' learning. Because the learning rate becomes smaller and smaller, it would take even longer to further reduce the estimation error.

However, since our learning algorithm can detect changes in the environment, it can increase the learning rate to learn the change quickly when an environmental change has been detected and decrease the learning rate with respect to equation 2.21 in order for the estimated mean to converge to the true mean when an environmental change has not been detected.

Specifically, TP/TPWG learning uses a big learning rate when an environmental change has been detected twice in a row and then decreases the learning rate according to equation 5.15 otherwise. In addition, as mentioned previously, we should use a smaller learning rate to learn the variance when the environment changes in order to reduce the influence of the environmental change on the estimated variance. After the mean is near its true value, we should use a bigger learning rate to learn the true variance quickly. If we use the same big learning rate (0.01) to learn variance whether the environment changes or not, the estimated variance may become so big that the changed actual time can still

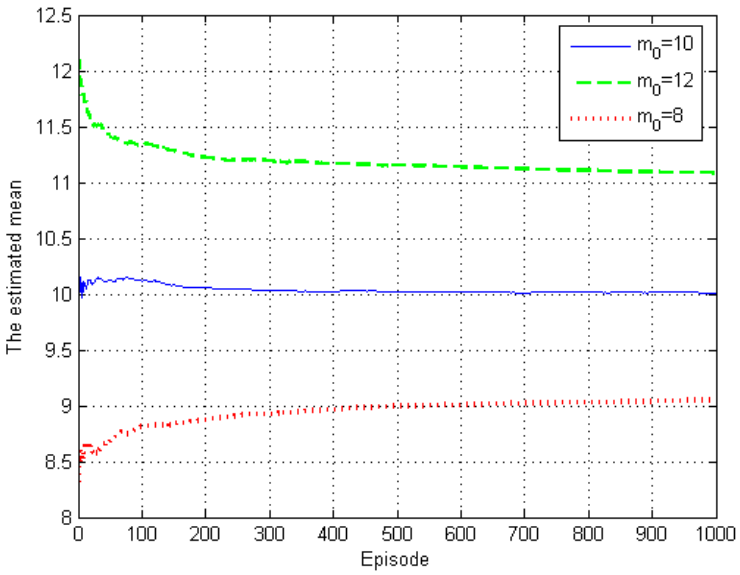

Figure 5.4: Learn the mean of a random variable with a decreasing variable learning rate; $\alpha(n) = \frac{\alpha(0)}{n}$ where n is the nth estimation and $\alpha(0) = 0.1$, the data is drawn from a Poisson distribution with a mean of 10; m_0 is the initial estimation of the mean. The learning is quite slow even in stationary environments if the initial estimated mean is not close to the true mean and the initial learning rate is not very big.

fall within the estimated mean plus and minus k standard deviations and therefore the learning rate used to learn the mean starts to decrease, even though the estimated mean is still far away from the true mean. As figure 5.5 shows, it would take the learning agent forever to learn the true mean. On the other hand, if we use the same small learning rate (0.001) to learn variance whether the environment changes or not, the variance is so small that the actual time may fall outside the estimated mean plus and minus k standard deviations even though the estimated mean is very close to the true mean and therefore the learning rate is increased incorrectly. As figure 5.6 shows, even when the mean has converged, α is still increased accidentally.

For these reasons, we use a smaller learning rate to learn variance when a change in the environment is detected and a greater learning rate if not. It is worth pointing out that even with this parameter setting, the above issues cannot be completely eliminated: the variance may still be incorrectly estimated with some data and this may further lead to incorrect estimation of the mean. However, the chance that the issues happen becomes much smaller. In the following experiment, we set $\alpha_2 = 0.002$ when an environmental change has been detected, $\alpha_2 = 0.01$ when it has not. As firgure 5.7 shows, the learning is not only fast but also stable.

In addition, as discussed previously, when the actual time to reward for a suboptimal action is not smaller than its estimated mean minus k standard deviations or is not smaller than the estimated time to reward for the optimal action in the state, it is likely that the suboptimal action is still suboptimal. Therefore, we decrease the exploration rate of the state (e.g. ϵ for ϵ-greedy) gradually towards its minimum value with equation 5.11 in order to reduce the cost of exploration of suboptimal actions and in order to enable the learning to converge to a better policy.

5.2.2.5 Evaluate whether or not the time information has been correctly learned

Although continuous exploration is needed in nonstationary environments, it is not necessary to continue this time's exploration of the current action or the current state-action pair beyond the time when the learning agent has found that the action is still worse than the optimal one. It is worth noting, however, that this does not affect future exploration of the action or the state-action pair. For

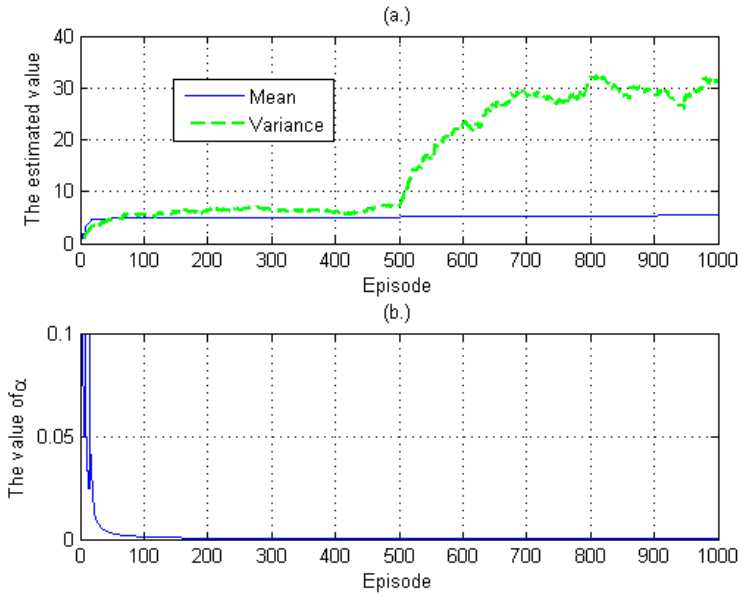

Figure 5.5: Learn the mean of a variable with a variable α and a fixed big α_2; $\alpha(n) = \alpha(0)$ if an environmental change is detected twice in a row, otherwise $\alpha(n) = \frac{\alpha(0)}{n+1}$ where n is the number that an environmental change has not been detected in a row, $\alpha(0) = 0.1$; $\alpha_2 = 0.01$; the data is drawn from a Poisson distribution with a mean of 6 in the first 500 episodes and then is drawn from a Poisson distribution with a mean of 10. If we use the same learning rate (0.01) to learn the variance whether the environment changes or not, the estimated variance may become so big when the environment changes that the changed actual time can still fall within the estimated mean plus and minus k standard deviations and therefore the learning rate used to learn the mean continues to decrease, even though the estimated mean is still far away from the true mean.

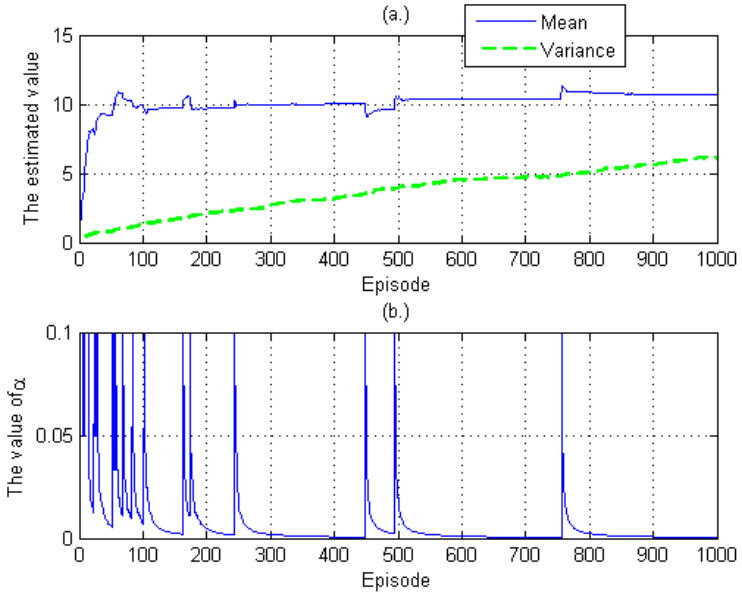

Figure 5.6: Learn the mean of a random variable with a variable α and a fixed small α_2; $\alpha(n) = \alpha(0)$ if the environmental change is detected twice in a row, otherwise $\alpha(n) = \frac{\alpha(0)}{n+1}$ where n is the number that the environmental change has not been detected in a row, $\alpha(0) = 0.1$; $\alpha_2 = 0.001$; the data is drawn from a Poisson distribution with a mean of 10. If we use the same small learning rate (0.001) to learn variance whether the environment changes or not, the variance is so small that the actual time may fall outside the estimated mean plus and minus k standard deviations even though the estimated mean is very close to the true mean and therefore the learning rate is increased incorrectly.

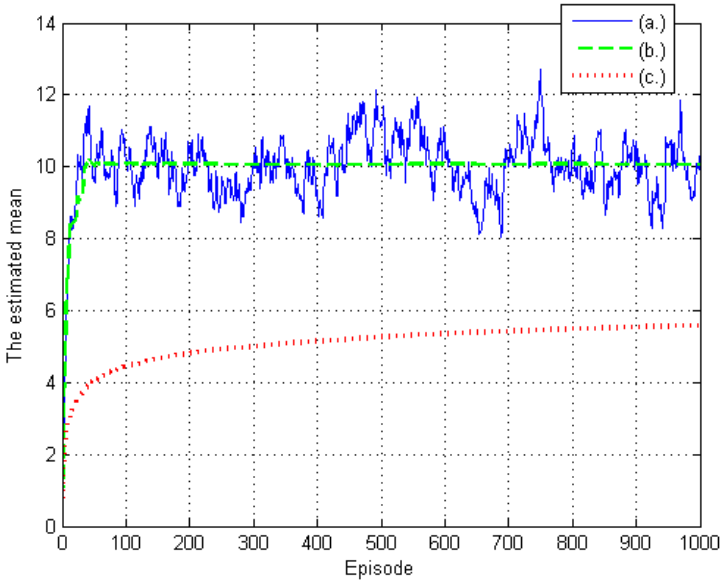

Figure 5.7: Comparison of learning the mean of a variable with different learning rates; the initial estimation is all 0; the data is drawn from a Poisson distribution with a mean of 10; (a.) a fixed learning rate (0.1); (b.) $\alpha(n) = \alpha(0)$ if the environmental change is detected twice in a row, otherwise $\alpha(n) = \frac{\alpha(0)}{n+1}$ where n is the number that the environmental change has not been detected in a row, $\alpha(0) = 0.1$; $\alpha_2 = 0.002$ when the environmental change has been detected, $\alpha_2 = 0.01$ when it has not. (c.) $\alpha(n) = \frac{\alpha(0)}{n}$ where n is the nth estimation and $\alpha(0) = 0.1$.

simplicity, we consider a time delayed 2-armed bandit problem. The two arms have the same amount of reward that does not change over time. The time to reward for the two arms is deterministic but nonstationary. Suppose that the agent has learned that it takes the agent 1 minute to get a reward if it chooses the first arm and 1 year to get the same amount of reward if it selects the second arm. Even so, the agent still needs to explore/select the second arm occasionally just in case the environment has changed and the time to reward for the second arm has become even shorter than that for the first arm, e.g. 1 second. This is the classical trade-off between exploration and exploitation. On the other hand, however, does the agent need to wait (1 year) until receiving a reward after it picks the second arm? Fortunately, this is not necessary. If the reward has not arrived 1 minute after the second arm is selected, the agent can conclude that the second arm is still worse than the first one, so the purpose of this time's exploration has been served and the agent does not need to wait there any longer. One important condition for this conclusion is that the estimated time to reward for the first arm, which is used to compare with the current action (the second arm), should not have been underestimated. Otherwise, the agent may even give up the optimal action.

When the actual time to reward for action a falls within $T(a) \pm k\sqrt{T\_var(a)}$, we consider that its time information has been correctly learned. Otherwise, we consider it not. Initially, we set the estimated variance to 0 so that it will not be evaluated as correct unless it is in a deterministic environment and the estimated mean is the correct one. When the actual time to reward for action a is no more than $T(a) + k\sqrt{T\_var(a)}$, we consider the estimated time to reward for action a has not been underestimated and action a has the correct time information for comparison. It can be used to decide whether to give up another action. This simple method is used in TP learning.

In order to make it robust to noise, we can make some modifications to the above simple rule. The prediction of whether the time information for one action has been learned correctly only changes when the action has been taken n times and its actual time to reward all falls within or outside $T(a) \pm k\sqrt{T\_var(a)}$ in a row. Obviously, when $n = 1$, it is just the above simple rule. Alternatively, we can also make the prediction probabilistic. When the actual time to reward for the action a falls within $T(a) \pm k\sqrt{T\_var(a)}$, the probability that the time information has been correctly learned increases towards 1; otherwise, it decreases

towards 0.

5.2.2.6 Give up the current action

Assume that the time to reward for the current action a has been correctly learned and the time to reward for at least one of the other actions (a') has not been underestimated. We use a'' to denote the action which has the minimum T value among all a'. If the estimated time to the reward for a is more than that for a'' and also the learning agent has spent in the current action (a) for more than the estimated time to reward for a'', the learning agent will give up the current action and then remake its decisions.

This is the simplest one, but it is not optimal when the actual time to reward for a is less than twice the estimated time to reward for a''. For example, suppose that the time to reward for a is 3 seconds and the time to reward for a'' is 2 seconds. If the agent waits for 2 seconds after choosing a, and then gives up and takes a'' instead, it will take the agent 4 seconds to get a reward. On the other hand, however, if the agent waits until a reward arrives after selecting a, it will only take the agent 3 seconds to get a reward. In this case, it is better not to give up when a is chosen because the agent can get the reward earlier by staying in a than by giving up and choosing a'' instead. In addition, in some special cases, this method may make the learning unstable. For instance, suppose that there are two arms. The time to reward for the first arm is deterministic and the time to reward for the second one is stochastic. The mean of the time to reward for the first arm is slightly less than the mean of the time to reward for the second one. The actual time to reward for the second arm may be less than the mean of the time to reward for the first arm in some episodes. If the learning agent gives up when the time elapsed after the second action is chosen exceeds the mean of the time to reward for the first arm, the estimated mean of the time to reward for the second arm is only updated in episodes when its actual time to reward is less than the mean of the time to reward for the first arm. Gradually, the estimated mean of the time to reward for the second arm will become less than or equal to that for the first arm, though the actual mean of the time to reward for the second arm is still greater than that for the first arm. After that, the second arm is chosen most of time and is not given up. The learning agent will soon find that the mean of its time to reward is actually still greater than that for the first arm, so it will then choose the first arm most of time and another cycle of switching

between arms commences.

The second method is that the agent will not give up unless the estimated time to reward for a is more than twice the estimated time to reward for a''. If this condition is satisfied, most of the time, the learning agent only explores $T(a'')$ after it takes a; But at times, it explores $2T(a'')$ after choosing a just in case the time to reward for a has become less than $2T(a'')$ where it is better not to give up when a is chosen. This method can solve the instability problem caused by the first method. For the previous example, when the estimated time to reward for the second arm becomes less than twice that for the first arm, the learning agent will not give up the second arm. Afterwards, the estimated time to reward for the second arm should increase and become more than twice that for the first arm statistically. Therefore, the estimated time to reward for the second arm is more than that for the first arm, viz. the first arm is the estimated optimal one, all the time. This method is used in TP learning.

Finally, in both of the above methods, the waiting time after action a is taken is proportional to the estimated mean of the time to reward for action a''. Instead of using $T(a'')$, we can also use $T(a'') + k\sqrt{T\_var(a'')}$ to decide when to give up. Obviously, this method encourages longer exploration because the agent explores suboptimal actions longer before giving up. Furthermore, the agent can also choose actions based on their estimated mean plus k standard deviations instead of their estimated mean alone in order to encourage exploration of actions whose values are uncertain. In this sense, it is similar to the Interval Estimation (IE) algorithm [69] where the action with the largest upper interval boundary is selected most of the time instead of the one with the largest mean.

It is worth pointing out that when the time information of the current action is not correct, it is also safe to give up. However, after giving up, the time to reward for the action cannot be correctly learned because the learning agent has not received the reward yet before giving up. Therefore, its time information cannot be used to decide whether to give up other actions later on. In addition, it cannot guarantee that the actual mean of the time to reward for a is more than twice the actual mean of the time to reward for a'' when the estimated time to reward for a is more than twice the estimated time to reward for a'' because the estimated time to reward for a may not be correct. Finally, if the learning agent is allowed to give up a even though the estimated time to reward for a is not more than the estimated time to reward for a'', we need to specifically increase

its estimation after giving up because its estimation cannot be updated through the normal updating rule due to the giving up. Otherwise, action a would stay as the optimal one and the learning agent would keep choosing action a most of time and then giving it up. One way to solve the problem is to update $T(a)$ and $T\_var(a)$ using an assumed time to reward. When the learning agent gives up at the t^{th} time step, it may expect the reward to be able to receive in kt time steps where $k > 1, k \in \Re$. Thus, we can update $T(a)$ and $T\_var(a)$ by using the assumed time to reward instead of the actual time to reward.

After giving up, the learning agent can remake its decision with respect to its policy. If so, however, the learning agent may still choose the previous action (a) after giving up. Although this helps to update the T value of this action, it may take the learning agent a long time to find whether or not the time to reward for the action (a'') used to decide to give up the previous action has also changed. If the time to reward for that action also becomes longer, the learning agent should not give up the current action since the time to reward for the compared action is not correct for comparison any more. Alternatively, the learning agent can just choose the action used to decide to give up the previous action in order to find whether or not the time to reward for that action has changed. If the time to reward for that action has become longer, it will not give up the action next time. If the time to reward for that action has not changed, however, this does not help the learning agent to recover from the environmental change because the T values of all actions have not changed at all and therefore their rank also does not change. This alternative method is used in TP learning after the learning agent gives up.

It is worth pointing out that the method of giving up used by TP learning is flexible and robust enough to cope with almost every case. Firstly, it always explores the current action at least the estimated time to reward for a'' just in case the time to reward for the current action (a) has become less than the estimated time to reward for a'' and a has become the optimal action. Secondly, when the estimated time to reward for the current arm is less than twice the estimated time to reward for a'', the learning agent does not give up the current action because it would take more time to get a reward if it gives up. On the other hand, when the estimated time to reward for the current arm is more than twice the estimated time to reward for a'', the learning agent would give up the current action at the time step equal to the estimated time to reward for a'' most of time. But just in

case the time to reward for a has become less than $2T(a'')$ where it is better not to give up when a is chosen, it explores $2T(a'')$ after a is taken occasionally. Given that the time information has been correctly learned, this method can behave near optimally in almost all settings of the time to reward.

5.2.3 Value (discounted reward) estimation

This algorithm is quite like algorithm 1. The only difference is that it learns the discounted reward and then uses it to make decisions instead of the time to reward. As shown in algorithm 5, a standard value (discounted reward) estimation reinforcement learning algorithm learns the discounted reward (Q) through an incremental updating rule with a fixed learning rate and then uses it to make decisions through a non-greedy rule, e.g. ϵ-greedy or softmax methods, with respect to the Q values of arms/actions. Since the n-armed bandit problem has only one state (but n actions) and we only consider it as a one-step episodic task, it has no state transition and Q values are only updated when a reward is received. In essence, this algorithm is just a simplified version of classical reinforcement learning algorithms to learn the discounted reward with only one state.

Algorithm 5 V learning

Inputs: Q; **Outputs:** Q; **Parameters:** α, ϵ, γ; **Internal variables:** t
for all episodes **do**
  Choose a_c from all possible actions using the policy derived from Q (ϵ-greedy) and then take action a_c
  $t = 0$
  repeat
    $t \leftarrow t + 1$
  until a reward r is received
  $Q(a_c) \leftarrow Q(a_c) + \alpha\left[\gamma^t r - Q(a_c)\right]$
end for

5.2.4 Value (discounted reward) estimation with value perception

TP/TPWG learning improves the standard time estimation reinforcement learning algorithm by learning both the estimated mean and variance of the time to reward, and then using them to detect any change in the environment and responding quickly to it if a change is detected. TP learning goes further by giving

up the current action if it has discovered that the current action is still worse than the optimal one. But they only work when the amount of reward for actions is the same and does not change. Firstly, the learned time information alone cannot be used to make sensible decisions if the amount of reward for actions is not the same. Secondly, it cannot detect changes in the amount of reward. If the amount of reward for actions is not the same but does not change, the agent can still use the learned time to reward to detect changes in the environment and then respond to the change quickly. Finally, the agent cannot just use the learned time to reward to decide when to give up because the giving up time is also decided by the relative amount of reward for actions in addition to the time to reward for actions. It is obvious that a bigger reward is worth waiting longer for. Therefore, we also need to learn the mean of the amount of reward in order to decide when to give up if the amount of reward for actions is not the same. For instance, consider a deterministic stationary environment for simplicity. Suppose that the reward for the current action has an amount of r_1, the reward for another action has an amount of r_2 occurred at t_2, and the time has elapsed t and no reward has been received since the learning agent chose the current action. Also suppose that the criterion of optimality is to maximise the discounted reward with a discounted factor γ. Only when $t > log_\gamma r_2 - log_\gamma r_1 + t_2$, viz. $\gamma^t r_1 < \gamma^{t_2} r_2$, the agent can deduct that the current action is worse than the compared action and therefore the learning agent can give up the action and end this time's exploration.

From the viewpoint of the learning agent, the learning agent will not consider the environment changed if its optimal target, e.g. the discounted reward, has not changed, though in reality the environment may have changed, e.g. the time to reward and the amount of reward both have changed. Regarding giving up, the learning agent should not give up unless the new amount of reward has been correctly learned when the amount of reward changes because the giving up time is also decided by the amount of reward as discussed above.

For the above reasons, we learn the mean (Q) and variance (Q_var) of the discounted reward and the mean (R) and variance (R_var) of the amount of reward. We then use the learned mean and variance of the discounted reward to detect environmental changes (from the viewpoint of the learning agent). When the actual time to reward t and the actual amount of reward r for the current

action a in the current episode satisfy

$$Q(a) - k\sqrt{Q\_var(a)} \leq \gamma^t r \leq Q(a) + k\sqrt{Q\_var(a)} \tag{5.16}$$

where γ is the discount rate, we consider the environment unchanged and $Q(a)$ has been correctly learned. Otherwise, we consider that the environment has changed and $Q(a)$ has not been correctly learned. If a change in the environment is detected, the learning rate is increased in order to learn the change quickly. Otherwise, the learning rate is decreased gradually towards 0 in order for the estimated mean of the discounted reward to converge to its true mean. If a suboptimal action has improved and can potentially become the optimal action in the new environment, the exploration rate (e.g. ϵ for ϵ-greedy) is also increased in order to increase the chance that the suboptimal action is visited. Otherwise, the exploration rate is decreased gradually towards its minimum value in order to reduce the cost of exploring suboptimal actions.

Likewise, for VP learning only, we use the learned mean and variance of the amount of reward to detect changes in the amount of reward and evaluate if the amount of reward has been correctly learned. When the actual amount of reward r of the current action a in the current episode satisfies

$$R(a) - k\sqrt{R\_var(a)} \leq r \leq R(a) + k\sqrt{R\_var(a)} \tag{5.17}$$

we consider the amount of reward unchanged and $R(a)$ has been correctly learned. Otherwise, we consider that the amount of reward has changed and $R(a)$ has not been correctly learned. When $R(a)$ has been correctly learned, the learning agent is allowed to give up the current action. Otherwise, it is not allowed to give up the current action. If $R(a)$ has not been correctly learned, $R(a)$ is either bigger or less than the actual mean of the amount of reward. When $R(a)$ is less than the actual mean of the amount of reward, the learning agent would give up too early if it is allowed to give up because the learning agent would wait a shorter time for a smaller amount of reward. When $R(a)$ is more than the actual mean of the amount of reward, the learning agent would give up too late if it is allowed to give up because the learning agent would wait longer for a greater reward. This is the reason why the learning agent is only allowed to give up the current action a when $R(a)$ has been correctly learned. In addition, even if $R(a)$ has been correctly learned, the learning agent should also occasionally explore the amount

of reward for action a, viz. not give up, just in case the amount of its reward has changed.

Assume the learning agent has spent in the current action a t time steps, $Q(a)$ and $R(a)$ has been correctly learned, and there exists at least one other action which has the correct Q values for comparison. We use a'' to represent the action which has the maximum Q value among those which have the correct Q values for comparison except a. If the conditions $\gamma^t R(a) < Q(a'')$, viz. $t > log_\gamma Q(a'') - log_\gamma R(a)$, and $T(a) > log_\gamma Q(a'') - log_\gamma R(a)$ are both satisfied, it is likely that the current action is still suboptimal and therefore the learning agent can give up the current action and then remake its decision. In order to avoid learning T, we use $log_\gamma \frac{Q(a)}{R(a)}$ to approximately replace $T(a)$, so we can get the condition $Q(a) < Q(a'')$.This method of calculating the timing of giving up, however, is not optimal when the actual time to reward for a is less than twice $log_\gamma Q(a'') - log_\gamma R(a)$. Similar to TP learning, it is better not to give up in this case. Thus, in the second method, the agent will not give up unless the expected time to reward for the current action $T(a)$ is more than $2(log_\gamma Q(a'') - log_\gamma R(a))$. Similar to the above, in order to avoid learning T, we use $log_\gamma \frac{Q(a)}{R(a)}$ to approximately replace $T(a)$, so we can get the condition $Q(a) < \frac{Q^2(a'')}{R(a)}$. If this condition is satisfied, most of the time, the learning agent only explores $log_\gamma Q(a'') - log_\gamma R(a)$ in a; But at times, it explores $2(log_\gamma Q(a'') - log_\gamma R(a))$ in a just in case the time to reward for a has become less than $2(log_\gamma Q(a'') - log_\gamma R(a))$ where it is better not to give up when a is chosen. Again, similar to TP learning, we can also encourage longer exploration by allowing the learning agent to give up when the time has exceeded $log_\gamma(Q(a'') - k\sqrt{Q\_var(a'')}) - log_\gamma R(a)$ instead of $log_\gamma Q(a'') - log_\gamma R(a)$. Furthermore, the agent can also choose actions based on their estimated mean plus k standard deviations instead of their estimated mean alone in order to encourage exploration of actions whose values are uncertain. Furthermore, the agent can also choose actions based on their estimated mean plus k standard deviations instead of their estimated mean alone in order to encourage exploration of actions whose values are uncertain. In this sense, it is similar to the Interval Estimation (IE) algorithm [69] where the action with the largest upper interval boundary is selected most of the time instead of the one with the largest mean.

As shown in algorithm 6, VP learning is quite similar to TP learning and VPWG learning is quite similar to TPWG learning. Most of the variations and

discussions regarding TP/TPWG learning are also applicable to these two algorithms except that it can handle the cases where the amount of reward for actions may be different and may also change.

Algorithm 6 VP/VPWG learning

Inputs: Q, Q_var, R, R_var; **Outputs:** Q, Q_var, R, R_var;
Parameters: k; **Internal variables:** A, a_c, t, ϵ (initial value: ϵ_{min}), $flag\_correctEst$ (initial value: $FALSE$), $count\_correct$ (initial value: 0), a_g (for VP learning only), $threshold$ (initial value: -∞, for VP learning only), $action\_forCmp$ (initial value: 0, for VP learning only), $flag\_correctCmp$ (initial value: $FALSE$, for VP learning only)
for all episode **do**
  Choose a_c from all possible actions using the policy derived from Q (ϵ-greedy); then take action a_c
  $t = 0$
  $a_g = 0$ {This line is for VP learning only}
  Use algorithm 8 to calculate $threshold(a_c)$ and $action\_forCmp(a_c)$ {This line is for VP learning only}
  repeat
    {Below is for VP learning only}
    if $a_g \neq 0$ **then**
      $a_c = a_g$, $a_g = 0$, $t = 0$
      Use algorithm 8 to calculate $threshold(a_c)$ and $action\_forCmp(a_c)$
    end if
    if $\gamma^t R(a_c) < Q(a_c) - k\sqrt{Q\_var(a_c)}$ **then**
      $flag\_correctEst(a_c) = FALSE$, $flag\_correctCmp(a_c) = FALSE$
    end if
    if $flag\_correctEst(a_c) = TRUE$ AND $flag\_correctEstR(a_c) = TRUE$ AND $\gamma^t R(a_c) < threshold(a_c)$ **then**
      $a_g = action\_forCmp(a_c)$ {Give up and then choose a_g}
    end if
    {Above is for VP learning only}
    $t \leftarrow t + 1$
  until a reward r is received
  Use algorithm 7 to update the model
end for

5.2.5 Other criteria of optimality

In the above four subsections, we improve the standard time or value estimation reinforcement learning algorithm by learning the mean and variance of the time to reward when the amount of reward is the same for all actions and does not

Algorithm 7 Update the model

Inputs: a, Q, Q_var, R (for VP learning only), R_var (for VP learning only), r, t, ϵ, $count\_correct$, $count\_correctR$ (for VP learning only), $flag\_correctEst$, $flag\_correctEstR$ (for VP learning only), $flag\_correctCmp$ (for VP learning only)

Outputs: Q, Q_var, R (for VP learning only), R_var (for VP learning only), ϵ, $count\_correct$, $count\_correctR$ (for VP learning only), $flag\_correctEst$, $flag\_correctEstR$ (for VP learning only), $flag\_correctCmp$ (for VP learning only)

Parameters: α_0, α_{2max}, α_{2min}, ϵ_{max}, ϵ_{min}, γ, ϕ, k, δ, η

Internal variables: α, α_R, α_2, α_{2R}, a^*, Q_{old}, R_{old} (for VP learning only)

$flag\_correctCmp(a) = \begin{cases} TRUE & \gamma^t r \geq Q(a) - k\sqrt{Q\_var(a)} \\ FALSE & \text{otherwise} \end{cases}$ {This line is for VP learning only}

if $Q(a) - k\sqrt{Q\_var(a)} \leq \gamma^t r \leq Q(a) + k\sqrt{Q\_var(a)}$ **then**
  $flag\_correctEst(a) = TRUE$, $\alpha_2(a) = \alpha_{2max}$, $count\_correct(a) \leftarrow count\_correct(a) + 1$
else
  $flag\_correctEst(a) = FALSE$, $\alpha_2(a) = \alpha_{2min}$
  if this happens twice in a row **then**
    $count\_correct(a) = 0$
  end if
end if

$\alpha(a) = \frac{\alpha_0}{(count\_correct(a)+1+\delta)^\eta}$

{Below is for VP learning only}

if $R(a) - k\sqrt{R\_var(a)} \leq \gamma^t r \leq R(a) + k\sqrt{R\_var(a)}$ **then**
  $flag\_correctEstR(a) = TRUE$, $\alpha_{2R}(a) = \alpha_{2max}$, $count\_correctR(a) \leftarrow count\_correctR(a) + 1$
else
  $flag\_correctEstR(a) = FALSE$, $\alpha_{2R}(a) = \alpha_{2min}$
  if this happens twice in a row **then**
    $count\_correctR(a) = 0$
  end if
end if

$\alpha_R(a) = \frac{\alpha_0}{(count\_correctR(a)+1+\delta)^\eta}$

{Above is for VP learning only}

if a is not the optimal action **then**

$\epsilon = \begin{cases} \epsilon + \phi\left[\epsilon_{max} - \epsilon\right] & \gamma^t r > Q(a) + k\sqrt{Q\_var(a)} \ AND \ \gamma^t r > Q(a^*) \\ \epsilon + \phi\left[\epsilon_{min} - \epsilon\right] & \text{otherwise} \end{cases}$

end if

{Update the estimation}

$Q_{old}(a) = Q(a), Q(a) \leftarrow Q(a) + \alpha(a)\left[\gamma^t r - Q(a)\right]$

$Q\_var(a) \leftarrow Q\_var(a) + \alpha_2(a)\{[\gamma^t r - Q_{old}(a)][\gamma^t r - Q(a)] - Q\_var(a)\}$

$R_{old}(a) = R(a), R(a) \leftarrow R(a) + \alpha_R(a)\left[r - R(a)\right]$ {This line is for VP learning only}

$R\_var(a) \leftarrow R\_var(a) + \alpha_{2R}(a)\{[\gamma^t r - R_{old}(a)][\gamma^t r - R(a)] - R\_var(a)\}$ {This line is for VP learning only}

Algorithm 8 Calculate when it should give up (used by algorithm 6); for VP learning only

Inputs: a, A, Q, R, t, $flag\_correctCmp$
Outputs: $threshold$, $action\_forCmp$
Parameters: ϵ_2, ϵ_3; **Internal variables:** $flag\_exploreAmount$, a', A', a''
if $\exists a' \in A \backslash a$ satisfying $flag\_correctCmp(a') = TRUE$ **then**
  use A' to represent the set of all a', $a'' = \arg\max_{a' \in A'} [Q(a')]$
  $action\_forCmp(a) = a''$
  $flag\_exploreAmount = \begin{cases} TRUE & \text{with probability } \epsilon_3/2 \\ FALSE & \text{with probability } 1 - \epsilon_3/2 \end{cases}$
  if $flag\_exploreAmount = TRUE$ OR $Q(a) \geq \frac{Q^2(a'')}{R(a)}$ **then**
    $threshold = -\infty$
  else
    $threshold = \begin{cases} Q(a'') & \text{with probability } 1 - \epsilon_2/2 \\ \frac{Q^2(a'')}{R(a)} & \text{with probability } \epsilon_2/2 \end{cases}$
  end if
else
  $action\_forCmp(a) = 0, threshold = -\infty$
end if

change over time, and learning the mean and variance of both the discounted and undiscounted reward when the amount of reward may be different and may also change. When the amount of reward is the same for all actions and does not change over time, the optimal target used is to minimise the expected time to reward (get a reward in the shortest time). When the amount of reward may be different and may also change, the optimal target used is to maximise the expected discounted reward/return (the infinite-horizon discounted model).

Their application, however, is not limited to these two criteria of optimality and the ideas can be applied to virtually any criterion of optimality. For instance, they can also be used where the optimal target is to maximise the expected rate of rewards (the average-reward model). In this case, the mean (P) and variance (P_var) of the rate of rewards and the mean (R) and variance (R_var) of the amount of reward should be learned instead. The agent then uses the learned mean and variance of the rate of rewards to detect environmental changes (from the viewpoint of the learning agent) and the learned mean and variance of the amount of reward to evaluate if the amount of reward has been correctly learned. If a change in the environment is detected, the learning rate is increased in order to learn the change quickly. Otherwise, the learning rate is decreased gradually

towards 0 in order for the estimated mean of the rate of rewards to converge to its true mean. If a suboptimal action has improved and can potentially become the optimal action in the new environment, the exploration rate (e.g. ϵ for ϵ-greedy) is also increased in order to increase the chance that the suboptimal action is visited. Otherwise, the exploration rate is decreased gradually towards its minimum value in order to reduce the cost of exploring suboptimal actions. When $R(a)$ has been correctly learned, the learning agent is allowed to give up the current action. Otherwise, it is not allowed to give up the current action. In addition, even if $R(a)$ has been correctly learned, the learning agent should also occasionally explore the amount of reward for action a, viz. not give up, just in case the amount of its reward has changed.

Assume t time steps have passed since the current action a is chosen, $P(a)$ and $R(a)$ have been correctly learned, and there exists at least one other action (a') which has the correct P value for comparison, viz. not overestimated. We use a'' to represent the action which has the maximum P value among all a'. If the conditions $R(a)/t < P(a'')$ and $P(a) < P(a'')$ are both satisfied, the learning agent will give up the current action and then remake its decision. Other methods for determining the timing of giving up in subsection 5.2.2 are also applicable here.

5.3 Experimental settings

In the following experiments, we only consider 2-armed bandit problems for simplicity. Specifically, the experimental scenario is as follows. There are two arms, each with a certain reward occurring some time after being pushed. Each episode ends when a reward is received. After some training, the time to reward or the amount of reward for one or both arms changes to another value, either smaller or bigger. The environment may be deterministic or stochastic. Experiments for deterministic environments are discussed in section 5.4 and experiments for stochastic environments are discussed in section 5.5.

Each experiment, except the experiments with random data, is run 100 times to reduce the influence of random noise. When we experiment on the algorithms with random data, we generate 100 random cases and each case is only run once. The data which has the same value for the two arms are discarded, because in this case there is no difference between either of the two arms. In addition, these

cases are generated in advance and then used for all algorithms to make the experimental comparison fair.

One criterion used here to evaluate the recovery speed from an environmental change is to see how long it takes the estimated optimal action to become correct for the new environment and stable after the environment changes when ϵ-greedy is used for decision-making. When the estimated optimal action becomes stable, the behaviour of the learning agent has become stable and therefore we can say that the learning has converged. When the estimated optimal action becomes correct for the new environment, the learning agent will be able to choose the actual optimal arm most of time (with the probability $1 - \frac{(n-1)\epsilon}{n}$) where n is the number of actions and therefore will behave correctly in the new environment. Thus, the recovery time used here is just the time of the last change of the estimated optimal action after the environment changes if the final estimated optimal action is correct for the new environment. On the other hand, if the final estimated optimal action at the end of one trial is not correct for the new environment, the recovery time is equal to all the time steps that the learning agent takes after the environment changes and before the trial ends. If softmax methods are used for non-greedy decision making, however, the behaviour of the learning agent will not become stable unless the (relative) T/Q values of all actions converge because the probability that one action is chosen depends on both its own T/Q value and the T/Q values of other actions. For simplicity, when the rank of the T/Q values of all actions becomes stable, we also consider that the behaviour of the learning agent has become stable since the action with the biggest T/Q value is chosen with the biggest probability, the action with the second biggest T/Q value is chosen with the second biggest probability and so on. Therefore, if softmax methods are used, we consider the learning agent has recovered from the environmental change when the rank of T/Q values becomes correct for the new environment and stable after the environment changes.

For the same reason, the criterion used here to evaluate the learning speed of algorithms during training is to see how long it takes for the estimated optimal action to become correct for the environment and stable afterwards after the learning begins when ϵ-greedy is used for decision making. When softmax methods are used, the criterion becomes how long it takes the rank of T/Q values to become correct and stable for the environment, after the training starts.

The settings of parameters used in the following experiments are as follows

Notation	Meaning
t_1	the actual time to reward for the first arm before the environment changes
t'_1	the actual time to reward for the first arm after the environment changes
t_2	the actual time to reward for the second arm before the environment changes
t'_2	the actual time to reward for the second arm after the environment changes
r_1	the actual amount of reward for the first arm before the environment changes
r'_1	the actual amount of reward for the first arm after the environment changes
r_2	the actual amount of reward for the second arm before the environment changes
r'_2	the actual amount of reward for the second arm after the environment changes
q_1	the actual discounted reward for the first arm before the environment changes
q'_1	the actual discounted reward for the first arm after the environment changes
q_2	the actual discounted reward for the second arm before the environment changes
q'_2	the actual discounted reward for the second arm after the environment changes

Table 5.2: Summary of notation used to describe experimental scenarios for the time delayed n-armed bandit problem

when applicable. $\alpha = 0.1, \alpha_0 = 1.2, \alpha_{2min} = 0.002, \alpha_{2max} = 0.01, \epsilon = 0.1, \epsilon_{min} = 0.1, \epsilon_{max} = 1, \epsilon_2 = 0.1, \epsilon_3 = 0.1, \phi = 0.1, \gamma = 0.9, k = 3, \delta = 3, \eta = 1$. Initialise $Q(a) = 0, Q\_var(a) = 0, R(a) = 0, R\_var(a) = 0, T(a) = 0, T\_var(a) = 0,$ $action\_forCmp(a) = 0$, $flag\_correctEst(a) = FALSE$, $flag\_correctCmp(a) = FALSE$, $count\_correct(a) = 0$ for all actions when applicable. Initialise $threshold(a) = \infty$ for TP learning and $threshold(a) = -\infty$ for VP learning.

In order to facilitate the description of the experimental scenarios, we introduce more notation as shown in table 5.2.

5.4 Deterministic environments

5.4.1 Introduction

In this section, we compare the performance of algorithms in terms of the learning speed, the recovery speed from environmental changes, and the agent's performance after the learning converges in deterministic environments where both the amount of reward and the time to reward for each arm are deterministic but may change over time.

We first consider cases where the two arms have the same amount of reward that does not change over time. The time to reward, however, may be different and may also change over time. Then we consider more general and complex scenarios, viz. cases where the two arms may have different amounts of reward and different time to reward, and both of them may also change over time.

5.4.2 When the amount of reward for actions is the same and does not change

We first compare the training process of the three algorithms. The following two cases are considered. It is worth noting that the two arms have the same amount of a single reward ($r_1 = r_2 = r'_1 = r'_2 = 1$) .

1. Case 1: $t_1 = 6, t_2 = 10, t_1 < t_2$
   The time to reward for the first action is less than that for the second action and therefore the first action is optimal. The difference in the time to reward between the two actions is relatively small.

2. Case 2: $t_1 = 6, t_2 = 100, t_1 << t_2$
   This is similar to Case 1: the first action is optimal. However, the difference in the time to reward between the two actions is quite great.

In Case 1, as figure 5.8 shows, both TP learning and TPWG learning learn the time to reward (T) faster than T learning due to a bigger learning rate when the estimated mean is not correct, which is the usual case at the beginning of learning. TP learning and TPWG learning learn almost at the same speed. At the learning stage, the estimated time to reward is usually incorrect, so giving up has not occurred. In addition, giving up has not occurred after learning either because the difference in the time to reward between the two actions is relatively

small. For this reason, there is also little difference among the three algorithms after learning.

When the difference in the time to reward between the two actions increases, however, giving up takes effect. In Case 2, as figure 5.9 shows, TP learning performs much better than T learning and TPWG learning after the learning converges in terms of the time steps taken to get one reward thanks to giving up when the difference in the time to reward between actions is great. After the learning converges, TP learning can get a reward in almost half of the time that T learning and TPWG learning need to get a reward and the time taken to get one reward by TP learning is also much less noisy than that by T learning and TPWG learning. The noisy error bars produced by T learning and TPWG learning are due to the variation in the time taken to get one reward. In one episode of one experiment, it may take 6 time steps to get a reward; in the same episode of another experiment, however, it may take 100 time steps to get a reward. For TP learning, however, it may either take 6 time steps or around 12 time steps on average to get a reward in one episode due to giving up, so the variation is less and therefore its error bar is less noisy.

As mentioned in section 5.3, we use the time steps taken for the estimated optimal action to become correct and stable to measure the learning speed of algorithms. In this sense, however, we cannot see the difference among the three algorithms from either figure 5.9 or 5.8. Therefore figure 5.10 is drawn. In the first scenario (the actual time to reward for the two arms is respectively 6 and 10), TP learning and TPWG learning learn faster than T learning due to a big learning rate when the estimated value is not correct, which is the usual case at the beginning of learning. In the second scenario (the actual time to reward for the two arms is respectively 6 and 100), however, there is little difference between TP/TPWG learning and T learning. This is because the difference in the time to reward between actions is quite big and therefore even a small learning rate can make the estimation of the time to reward for the two arms differentiable in one or two episodes. It is worth noting, however, though it takes all three algorithms almost the same time steps to converge in the second case, the agent's performance after the learning converges is quite different between T/TPWG learning and TP learning as discussed above.

Then, we compare their performance in terms of recovering from environmental changes. We first consider the following typical scenarios. Initially, the two

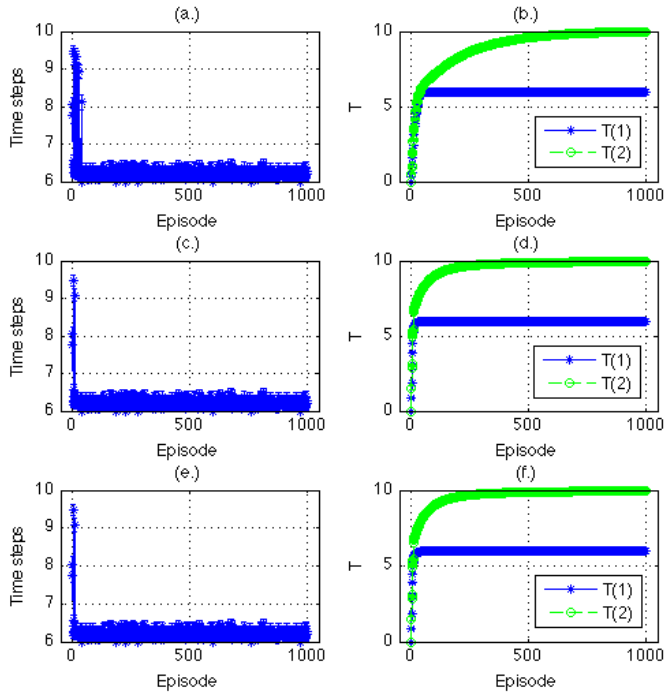

Figure 5.8: Comparison of the three algorithms during training; the time to reward for the two arms is respectively 6 and 10 time steps; the amount of reward for the two arms is both 1. The error bar in the graph is the sampling error, which is the case for all the following graphs. (a.) time steps taken to get the Xth reward by T learning; (b.) the learning of the estimated time to reward by T learning; (c.) time steps taken to get the Xth reward by TPWG learning; (d.) the learning of the estimated time to reward by TPWG learning; (e.) time steps taken to get the Xth reward by TP learning; (f.) the learning of the estimated time to reward by TP learning.

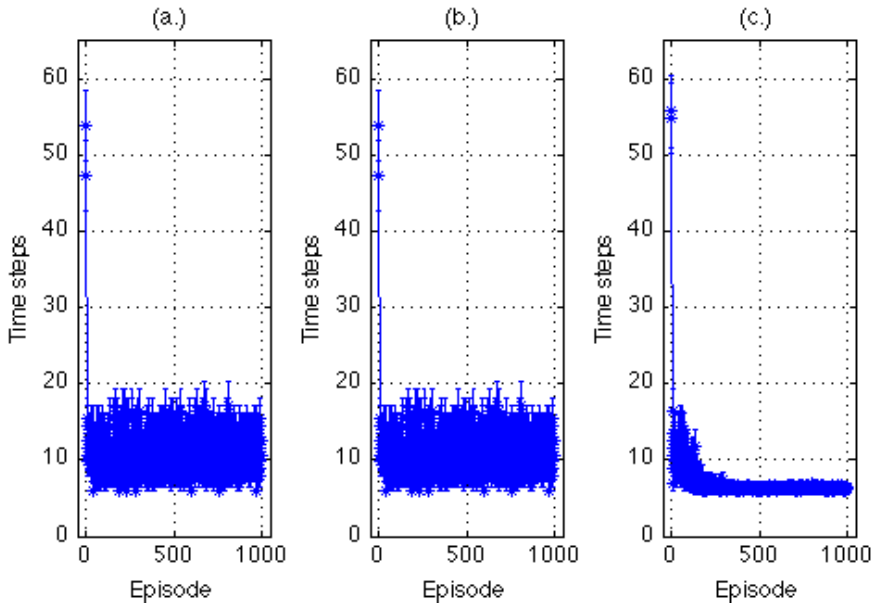

Figure 5.9: Comparison of the three algorithms during training; the time to reward for the two arms is respectively 6 and 100 time steps; the amount of reward for the two arms is both 1. (a.) time steps taken to get the Xth reward by T learning; (b.) time steps taken to get the Xth reward by TPWG learning; (c.) time steps taken to get the Xth reward by TP learning.

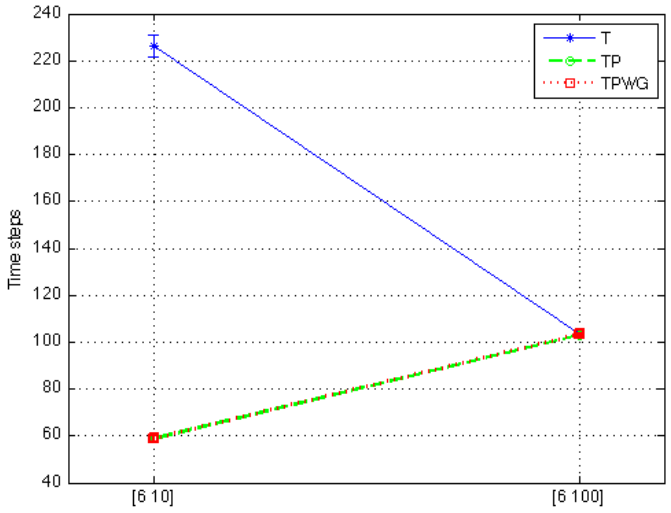

Figure 5.10: Comparison of the three algorithms during training in terms of time steps taken for the behaviour of the learning agent to become correct and stable for the environment, viz. successfully identifying the optimal arm and choosing it most of the time if ϵ-greedy is used to make decisions; the amount of reward for the two arms is both 1.

arms have the same amount of a single reward ($r_1 = r_2 = 1$) occurred respectively at the 6th and 10th time steps after being pushed ($t_1 = 6, t_2 = 10, t_1 < t_2$, the first arm is the optimal choice). After 1000 episodes, the time to reward for the two actions changes, though the amount of reward stays the same ($r'_1 = r'_2 = 1$).

1. Case 3: $t'_1 = t_1, t'_2 = 2 < t_2, t'_2 < t'_1$
   The time to reward for the first action does not change. The new time to reward for the second action becomes shorter: the previous suboptimal action (the second action) has become optimal.

2. Case 4: $t'_1 = 14 > t_1, t'_2 = t_2, t'_2 < t'_1$
   The new time to reward for the first action becomes longer: the previous optimal action (the first action) has become suboptimal. The time to reward for the second action does not change.

3. Case 5: $t'_1 = 14 > t_1, t'_2 = 18 > t_2, t'_1 < t'_2$
   The new time to reward for both actions becomes longer. In the new environment, the first arm is still the optimal one. Even though the rank of the actual time to reward for the two actions has not changed, the rank of the estimated time to reward by the learning agent will change over time. At the beginning, the estimated time to reward for the first action is smaller, so it is chosen more often and its estimated time to reward therefore increases more quickly and eventually becomes bigger than that for the second one. Then, the second action is chosen more often, so its estimated time to reward increases quicker and eventually becomes more than that for the first one. The process will go on until the estimated time to reward for at least one action converges.

4. Case 6: $t'_1 = 18 > t_1, t'_2 = 14 > t_2, t'_2 < t'_1$
   The new time to reward for both actions becomes longer. In the new environment, the first arm has become the suboptimal one and the second arm has become the optimal one.

As figure 5.11 shows, TP/TPWG learning recovers from environmental changes at least twice as fast as T learning in all experimental scenarios. In Case 3, TP/TPWG learning is more than 4 times faster in recovering from environmental changes than T learning. When the time to reward changes from [6 10] to

[6 2], increasing ϵ when suboptimal actions may have potentially become the optimal action has helped TP/TPWG learning to explore the second action. In all experimental scenarios, increasing the learning rate when the environment changes has accelerated the learning speed. It is worth noting that TP learning and TPWG learning perform almost the same in terms of the speed in recovering from environmental changes in all experimental cases. This is because giving up has not contributed to the recovery of TP learning from environmental changes. When the environment changes, the estimated time to reward becomes incorrect and therefore giving up has not occurred. TP learning, in this case, becomes equivalent to TPWG learning in essence.

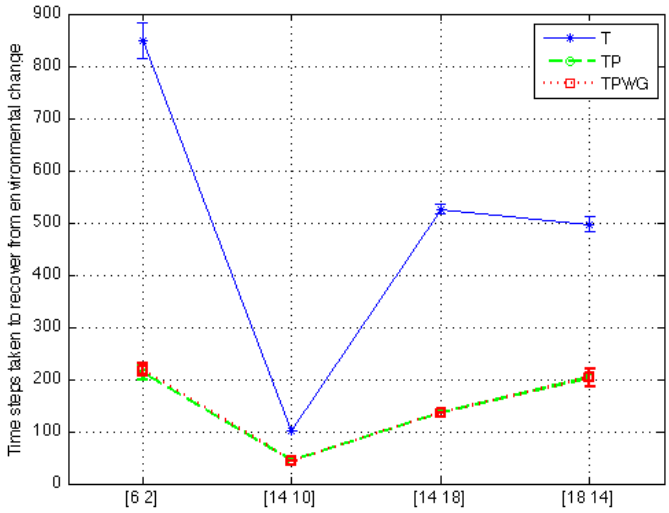

Figure 5.11: Time steps taken to recover from environmental changes after the time to reward changes from (6,10); the amount of reward for both arms (1) does not change.

In order to get the average performance of the algorithms, we also generate the time to reward for both actions randomly from integer number discrete uniform distributions at the first episode and the 1001th episode. The amount of reward for both arms is 1 all the time.

1. Case 7: $t_1, t_2, t_1', t_2' \sim IntegerDiscreteUniform(0, 100)$
   The time to reward for both arms is drawn from an integer number discrete uniform distribution ranging from 0 to 100 once at the beginning and once when the environment changes. In this scenario, the time to reward for both arms may become shorter, longer or stay the same (even though the

chance of which is very small). The rank of both arms may also stay the same or change when the environment changes.

2. Case 8: $t_1, t_1', t_2' \sim IntegerDiscreteUniform(0, 50)$;
   $t_2 \sim IntegerDiscreteUniform(51, 101)$
   The time to reward for the first arm is drawn from an integer number discrete uniform distribution ranging from 0 to 50 once at the beginning and once when the environment changes; the time to reward for the second arm is drawn from an integer number discrete uniform distribution ranging from 51 to 101 once at the beginning and from an integer number discrete uniform distribution ranging from 0 to 50 once when the environment changes. In this scenario, the time to reward for the first arm may become shorter, longer or stay the same (even though the chance of which is very small). But the time to reward for the second arm definitely decreases. The rank of both arms may stay the same or change from a certain rank (the first arm is optimal at the beginning) when the environment changes.

3. Case 9: $t_1 \sim IntegerDiscreteUniform(0, 50)$;
   $t_2, t_1', t_2' \sim IntegerDiscreteUniform(51, 101)$
   The time to reward for the first arm is drawn from an integer number discrete uniform distribution ranging from 0 to 50 once at the beginning and from an integer number discrete uniform distribution ranging from 51 to 100 once when the environment changes; the time to reward for the second arm is drawn from an integer number discrete uniform distribution ranging from 51 to 101 once at the beginning and once when the environment changes. In this scenario, the time to reward for the second arm may become shorter, longer or stay the same (even though the chance of which is very small). But the time to reward for the first arm definitely increases. The rank of both arms may also stay the same or change from a certain rank (the first arm is optimal at the beginning) when the environment changes.

4. Case 10: $t_1, t_2' \sim IntegerDiscreteUniform(0, 50)$;
   $t_2, t_1' \sim IntegerDiscreteUniform(51, 101)$
   The time to reward for the first arm is drawn from an integer number discrete uniform distribution ranging from 0 to 50 once at the beginning and from an integer number discrete uniform distribution ranging from 51 to 100 once when the environment changes; the time to reward for the second arm

is drawn from an integer number discrete uniform distribution ranging from 51 to 101 once at the beginning and from an integer number discrete uniform distribution ranging from 0 to 50 once when the environment changes. In this scenario, the time to reward for the first arm increases and the time to reward for the second arm decreases. The rank of both arms changes from a certain rank to another certain rank when the environment changes.

As figure 5.12 shows, TP/TPWG learning recovers from environmental changes at least twice as fast as T learning in all experimental scenarios. The difference between TP learning and TPWG learning is relatively small compared with their difference with T learning.

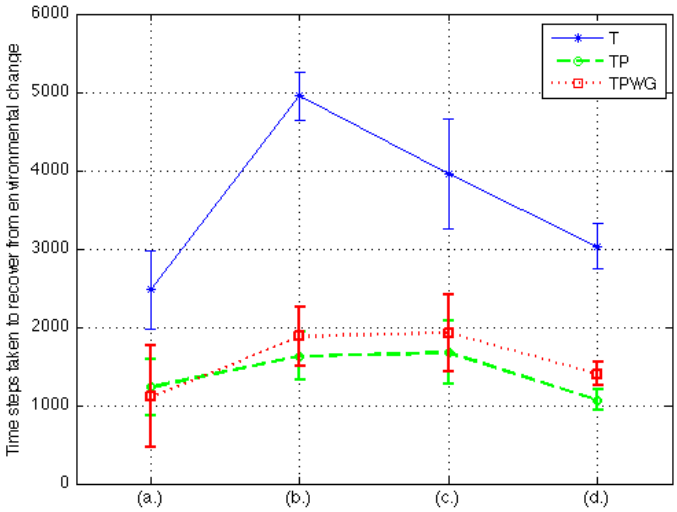

Figure 5.12: Time steps taken to recover from environmental changes after the time to reward changes; the amount of reward for both arms (1) does not change. (a.) Case 7; (b.) Case 8; (c.) Case 9; (d.) Case 10.

We then use Case 7 as a test bed to experiment on T learning and TP/TPWG learning with different initial learning rate (α_0). The results are shown in 5.13. It takes the learning agent fewer time steps to recover from environmental changes with a bigger initial learning rate, though the improvement over a smaller learning rate becomes smaller and smaller with the increase in the learning rate.

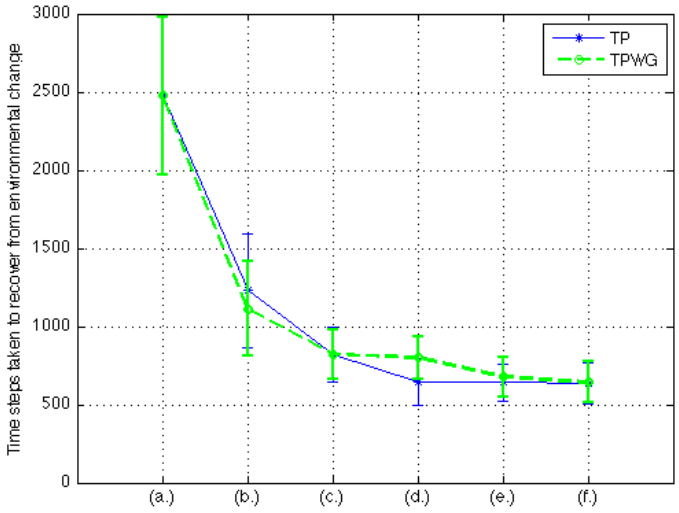

Figure 5.13: Time steps taken to recover from environmental changes with different initial learning rates; the time to reward for both arms is drawn from an integer number discrete uniform distribution ranging from 0 to 100 once at the beginning and once when the environment changes; the amount of reward for both arms (1) does not change. (a.) T learning ($\alpha = 0.1$); (b.) TP/TPWG learning with $\alpha_0 = 1.2$; (c.) TP/TPWG learning with $\alpha_0 = 1.6$; (d.) TP/TPWG learning with $\alpha_0 = 2.0$; (e.) TP/TPWG learning with $\alpha_0 = 2.4$; (f.) TP/TPWG learning with $\alpha_0 = 2.8$.

5.4.3 When the amount of reward for actions may be different and may also change

As discussed above, when the amount of reward for actions is not the same and may also change, learning the time to reward alone cannot make sensible decisions, detect changes in the amount of reward, or decide when to give up the current action. Therefore, in this subsection, we use V, VP and VPWG learning instead.

We first compare the training process of the three algorithms. The following two cases are considered.

1. Case 1: $t_1 = 6, r_1 = 6, t_2 = 10, r_2 = 10$
   The second action is optimal in terms of the discounted reward, but the difference between two actions is relatively small.

2. Case 2: $t_1 = 6, r_1 = 100, t_2 = 10, r_2 = 10$
   The first action is optimal in terms of the discounted reward, and the difference between two actions is relatively large.

In Case 1, as figure 5.14 shows, VP/VPWG learning learns much faster than V learning in terms of both the learning process and the learning of Q values due to a greater learning rate when the estimated value is not correct, which is the usual case at the beginning of learning.

Because the difference in Q values between the two actions is relatively small, giving up has not occurred after learning and therefore there is little difference between the three algorithms after learning. When the difference in the Q value between the two actions becomes greater, however, giving up takes effect. As figure 5.15 shows, VP learning performs better than V/VPWG learning after the learning converges in terms of the discounted reward received thanks to giving up when the difference in the Q value between actions is large.

Then, we compare their performance in terms of recovering from environmental changes. We consider the following two scenarios.

1. Case 3: $t_1, t_2, t_1', t_2' \sim IntegerDiscreteUniform(0, 100)$;
   $q_1, q_2, q_1', q_2' \sim RealNumberUniform(0, 100)$
   The time to reward for both arms is independently drawn from an integer number discrete uniform distribution from 0 to 100 inclusive once at the beginning and once when the environment changes, the value of discounted reward for both arms is independently drawn from a real number continuous

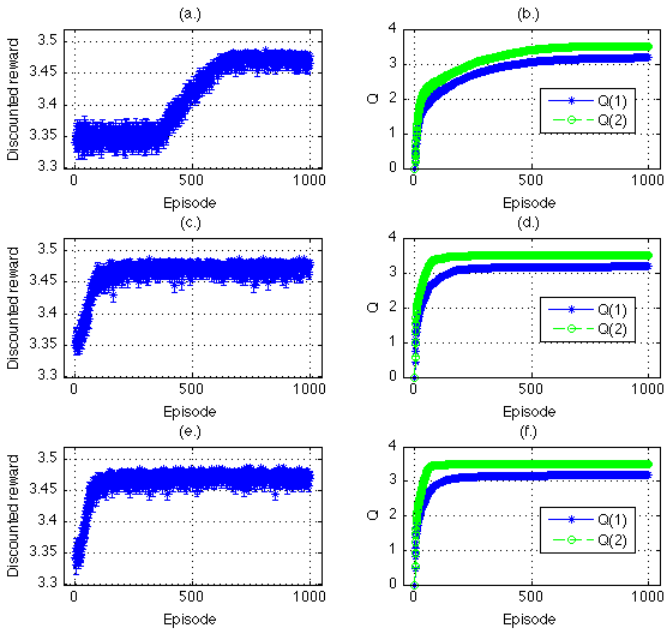

Figure 5.14: Comparison of the three algorithms during training; Case 1. (a.) discounted reward received by V learning; (b.) the learning of Q values by V learning; (c.) discounted reward received by VPWG learning; (d.) the learning of Q values by VPWG learning; (e.) discounted reward received by VP learning; (f.) the learning of Q values by VP learning.

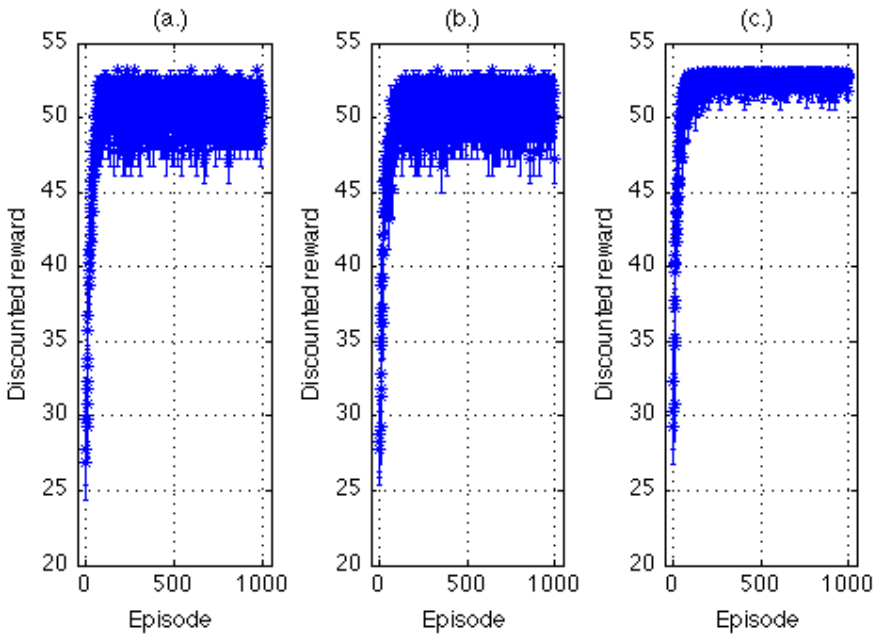

Figure 5.15: Comparison of the three algorithms during training; Case 2. (a.) discounted reward received by V learning; (b.) discounted reward received by VPWG learning; (c.) discounted reward received by VP learning.

uniform distribution from 0 to 100 exclusive once at the beginning and once when the environment changes, and the amount of reward is deduced from them. The reason why we generate the discounted reward rather than the amount of reward for experiments is that the discounted reward decides the optimal action and we use the last change of the optimal action to measure the recovery time.

2. Case 4: $t_1 = 6, r_1 = 10, t_2 = 10, r_2 = 100; t_1' = t_1, r_1' = 100 > r_1, t_2' = t_2, r_2' = r_2$
   The amount of reward for the first action increases and the first action becomes the optimal action in the new environment in place of the second one.

The first experiment tests their average performance. As figure 5.16 shows, both VP learning and VPWG learning recover from environmental changes more than twice as quickly as V learning. It is worth noting that VPWG learning performs even better than VP learning. In this experiment, the amount of reward for actions before and after the environment changes is generated randomly, so in some scenarios the amount of reward for the suboptimal action may increase

and the suboptimal action may become the optimal one. VP learning assumes that the amount of reward for suboptimal actions stays the same and therefore may still give up the suboptimal action even though it may have become the optimal one. If VP learning gives up the suboptimal action before receiving any reward, it cannot discover that the amount of reward for the suboptimal action has changed and the suboptimal action has become the optimal one. Even though with probability $\frac{\epsilon_3}{2}$, VP learning does not give up and therefore can discover the change in the amount of reward for the suboptimal action eventually, it would take VP learning much longer to find the change and to discover that the suboptimal action has become the optimal one than V/VPWG learning. To demonstrate this, we specifically design such a scenario, viz. Case 4. The results are shown in figure 5.17. VP learning with standard parameter settings performs the least effectively. With the increase in ϵ_3 (the probability of exploring the amount of reward, viz. not giving up, increases), the performance of VP learning is improved. VPWG learning performs best.

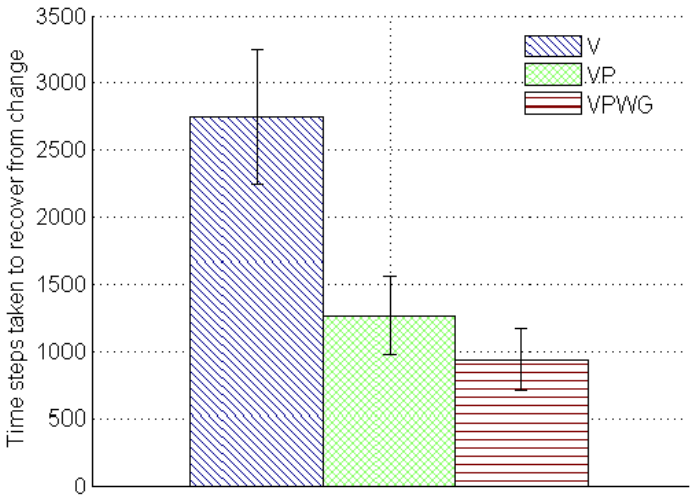

Figure 5.16: Time steps needed to recover from environmental changes after both the time to reward and the amount of reward change; Case 3.

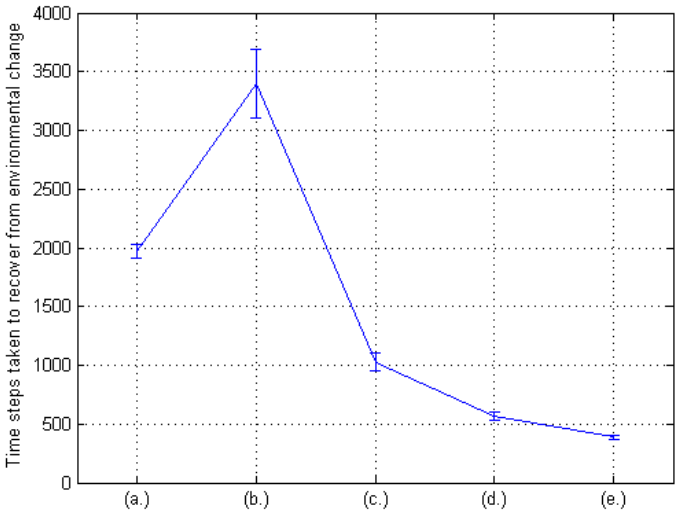

Figure 5.17: Time steps needed to recover from environmental changes; Case 4. (a.) V learning; (b.) VP learning, $\epsilon_3 = 0.1$; (c.) VP learning, $\epsilon_3 = 0.5$; (d.) VP learning, $\epsilon_3 = 1$; (e.) VPWG learning.

5.5 Stochastic environments

5.5.1 Introduction

In this section, we compare the performance of algorithms in terms of the learning speed, the recovery speed from environmental changes, and the agent's performance after the learning converges in stochastic environments where the amount of reward and the time to reward for each arm are stochastic and their distributions may also change over time. Stochastic environments make the problem more complex. Even if the distribution does not change over time, the actual time to reward or the actual amount of reward may still be different in different episodes.

We first consider cases where the two arms have the same deterministic amount of reward that does not change over time. The time to reward, however, is stochastic and its distribution may also change over time. Then we move on to more general and complex scenarios, viz. both the amount of reward and the time to reward for arms are stochastic, and both of their distributions may change over time.

5.5.2 When the amount of reward for actions is the same and does not change

In this subsection, we carry out a set of experiments similar to those in subsection 5.4.2. The difference is that the actual time to reward here is drawn from a Poisson distribution rather than being deterministic. Unlike deterministic environments, even though the mean of the actual time to reward is the same in different episodes of one experiment, the actual time to reward in each episode may be quite different.

We first compare the training process of the three algorithms. The following two cases are considered. It is worth noting that the two arms have the same amount of a single reward ($r_1 = r_2 = r'_1 = r'_2 = 1$) .

1. Case 1: $t_1 \sim Poisson(6), t_2 \sim Poisson(10), mean(t_1) < mean(t_2)$
   The mean of the time to reward for the first action is less than that for the second action, so the first action is optimal on average. The difference in the mean of the time to reward between the two actions is relatively small. It is worth pointing out, however, that these conclusions only apply to the average case. In some episodes, the situations can be quite different. For example, the actual time to reward for the first action in some episodes can be more than that for the second action and the difference in the actual time to reward between the two actions in some episodes can be very large.

2. Case 2: $t_1 \sim Poisson(6), t_2 \sim Poisson(100), mean(t_1) << mean(t_2)$
   This is similar to Case 1: the first action is optimal on average. However, the difference in the mean of the time to reward between the two actions is quite large.

The results are similar to those in the deterministic experiments. In Case 1, as figure 5.18 shows, TP/TPWG learning learns the estimated time to reward (T) faster than T learning due to a greater learning rate when the estimated value is not correct, which is the usual case at the beginning of learning. The difference among the three algorithms after learning is still very small. When the difference in the time to reward between the two actions increases, however, giving up takes effect for the same reason in the deterministic experiments. In Case 2, as figure 5.19 shows, TP learning performs much better than T/TPWG learning after the learning converges in terms of the time steps taken to get one reward

thanks to giving up when the difference in the time to reward between actions is great. In addition, as in the deterministic experiments, we also measure how many time steps taken for the behaviour of the learning agent to become correct and stable for the environment. The results are shown in figure 5.20 and they are quite similar to those in the deterministic experiments. In the first scenario (the mean of the actual time to reward for the two arms is respectively 6 and 10), TP/TPWG learning learns faster than T learning due to a faster learning rate when the estimated value is not correct, which is usually the case at the beginning of learning. In the second scenario (the mean of the actual time to reward for the two arms is respectively 6 and 100), however, there is little difference among the three algorithms. This is because the difference in the time to reward between actions is quite large. For all three algorithms, the learning agent only needs one or two episodes to make the estimated optimal action correct and stable for the environment. Similar to the results for the deterministic experiments, though it takes all three algorithms almost the same time steps to converge in the second case, the agent's performance after the learning converges is quite different between T/TPWG learning and TP learning as discussed above.

As mentioned in the description of the algorithms, TP/TPWG learning decreases the learning rate in order for the estimated mean to converge to the true mean when environmental changes have not been detected. Furthermore, when the mean of the time to reward for two arms is very close, algorithms which do not decrease the learning rate may struggle to find the optimal action in stochastic environments. Thus, we experiment on the algorithms using two scenarios where the actual mean of the time to reward for the two arms is very close.

1. Case 3: $t_1 \sim Poisson(14), t_2 \sim Poisson(16), mean(t_1) < mean(t_2)$
   In this scenario, the time to reward for the two arms is drawn respectively from Poisson(14) and from Poisson(16). The mean of the time to reward for the two arms is quite close.

2. Case 4: $t_1 \sim Poisson(15), t_2 \sim Poisson(16), mean(t_1) < mean(t_2)$
   In this scenario, the time to reward for the two arms is drawn respectively from Poisson(15) and from Poisson(16). The mean of the time to reward for the two arms is even closer.

As figure 5.21 shows, the estimated time to reward (T) by T learning fluctuates and therefore T learning struggles to find the optimal action in both scenarios.

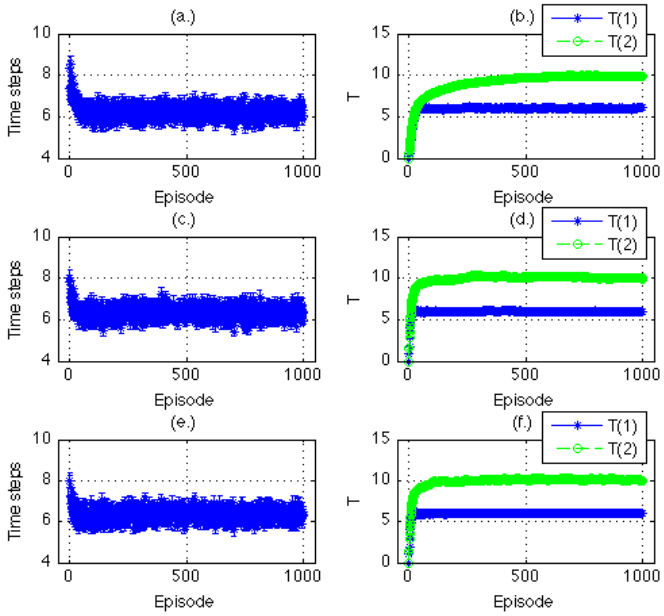

Figure 5.18: Comparison of the three algorithms during training; the time to reward for the two arms is drawn from Poisson(6) and Poisson(10); the amount of reward for the two arms is both 1. (a.) time steps taken to get the Xth reward by T learning; (b.) the learning of the time to reward by T learning; (c.) time steps taken to get the Xth reward by TPWG learning; (d.) the learning of the time to reward by TPWG learning; (e.) time steps taken to get the Xth reward by TP learning; (f.) the learning of the time to reward by TP learning.

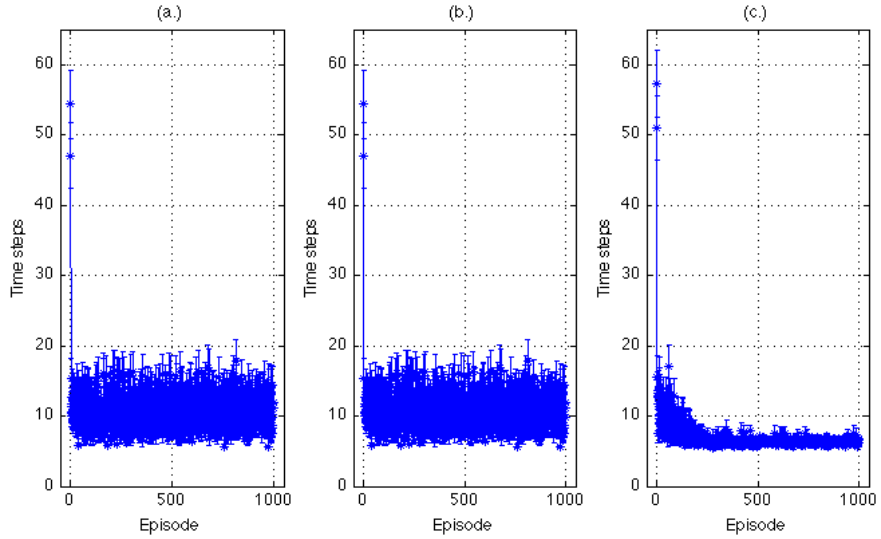

Figure 5.19: Comparison of the three algorithms during training, the time to reward for the two arms is drawn from Poisson(6) and Poisson(100); the amount of reward for the two arms is both 1. (a.) time steps taken to get the Xth reward by T learning; (b.) time steps taken to get the Xth reward by TPWG learning; (c.) time steps taken to get the Xth reward by TP learning.

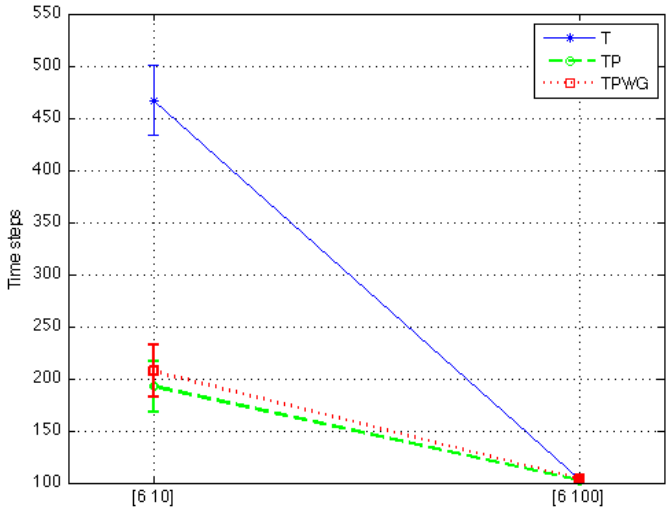

Figure 5.20: Comparison of the three algorithms during training in terms of time steps taken for the behaviour of the learning agent to become correct and stable for the environment; the amount of reward for the two arms is both 1.

The estimation estimated time to reward (T) by TP/TPWG learning, on the other hand, converges to their actual value and therefore TP/TPWG learning is able to find the optimal action in both scenarios.

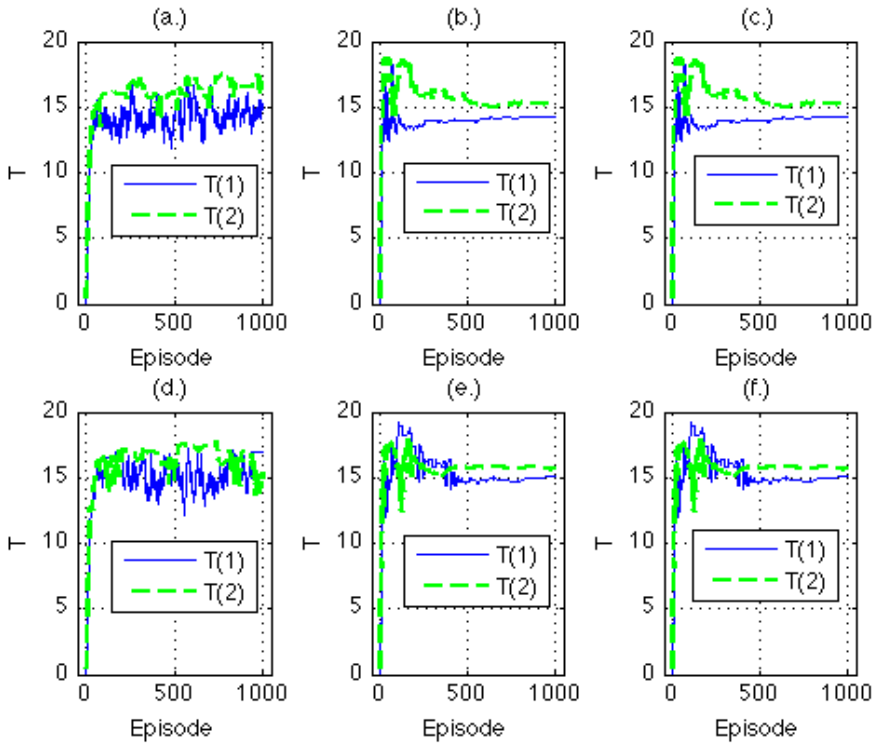

Figure 5.21: Comparison of the three algorithms during training in a typical run when the actual mean of the time to reward for two arms is very close; the time to reward for the two arms is drawn from Poisson(14) and Poisson(16) for (a.), (b.) and (c.); the time to reward for the two arms is drawn from Poisson(15) and Poisson(16) for (d.), (e.) and (f.); the amount of reward for the two arms is both 1. (a.) and (d.): the learning of T by T learning in a typical run; (b.) and (e.): the learning of T by TPWG learning in a typical run; (c.) and (f.): the learning of T by TP learning in a typical run.

Then, we compare their performance in terms of recovering from environmental changes. We first experiment on the alorithms using the following typical scenarios similar to the scenarios in the deterministic environments. Unlike the deterministic environments, however, the rank of actions in each episode is still uncertain even though the rank of actions on average is certain. Initially, the two arms have the same amount of a single reward ($r_1 = r_2 = 1$). The time to reward for actions is drawn from Poisson distributions with mean equal to respectively 6 and 10 time steps after being pushed ($t_1 \sim Poisson(6), t_2 \sim$

$Poisson(10), mean(t_1) < mean(t_2)$). After 1000 episodes, the mean of the time to reward for the two actions changes, though it is still drawn from a Poisson distribution. Furthermore, the amount stays the same ($r'_1 = r'_2 = 1$).

1. Case 5: $mean(t'_1) = mean(t_1)$, $t'_2 \sim Poisson(2)$; $mean(t'_2) < mean(t_2)$, $mean(t'_2) < mean(t'_1)$
   The new time to reward for the first action is drawn from the same distribution. The new time to reward for the second action is drawn from a Poisson distribution with a smaller mean: the suboptimal action (the second action) becomes optimal on average in the new environment. It is worth noting, however, that it is possible that the actual time to reward for the first arm is still less than that for the second arm in some episodes after the environmental change thanks to the stochastic environment.

2. Case 6: $t'_1 \sim Poisson(14)$, $mean(t'_2) = mean(t_2)$; $mean(t'_1) > mean(t_1)$, $mean(t'_2) < mean(t'_1)$
   The new time to reward for the first action is drawn from a Poisson distribution with a higher mean: the previous optimal action (the first action) becomes suboptimal on average in the new environment. The new time to reward for the second action is drawn from the same distribution.

3. Case 7: $t'_1 \sim Poisson(14)$, $t'_2 \sim Poisson(18)$; $mean(t'_1) > mean(t_1)$, $mean(t'_2) > mean(t_2)$, $mean(t'_1) < mean(t'_2)$
   The mean of the new time to reward for both actions increases. In the new environment, the first arm is still the optimal one on average. Similar to the case in the deterministic experiments, even though the rank of the mean of the actual time to reward for the two actions has not changed, the rank of the estimated mean of the time to reward by the learning agent will change over time. At the beginning, the estimated mean of the time to reward for the first action is smaller, so it is chosen more often and its estimated mean of the time to reward therefore increases more quickly and eventually becomes bigger than that for the second one. Then, the second action is chosen more often, so its estimated mean of the time to reward increases quicker and eventually becomes more than that for the first one. The process will go on until the estimated mean of the time to reward for at least one action converges.

4. Case 8: $t'_1 \sim Poisson(18)$, $t'_2 \sim Poisson(14)$; $mean(t'_1) > mean(t_1)$, $mean(t'_2) > mean(t_2)$, $mean(t'_2) < mean(t'_1)$
   Like the last case, the mean of the time to reward for both actions increases. Unlike the last case, however, the second arm, the previous suboptimal arm, becomes optimal on average in the new environment.

As before, the criterion used here to evaluate the performance of recovering from environmental changes is to see how long it takes before the estimated optimal action becomes correct for the new environment and becomes stable after the environment changes. As figure 5.22 shows, it takes TP/TPWG learning fewer time steps than T learning to recover from environmental changes in all scenarios similar to the results for deterministic environments. The smallest difference (around half) in the time steps taken to recover from environmental changes between TP/TPWG learning and T learning happens in Case 6. The difference in the other three cases is much greater. Similar to the results for deterministic environments, TP learning and TPWG learning perform almost the same in terms of the speed in recovering from environmental changes for similar reasons. It is also worth noting that it takes all three algorithms more time to recover from environmental changes than in deterministic environments.

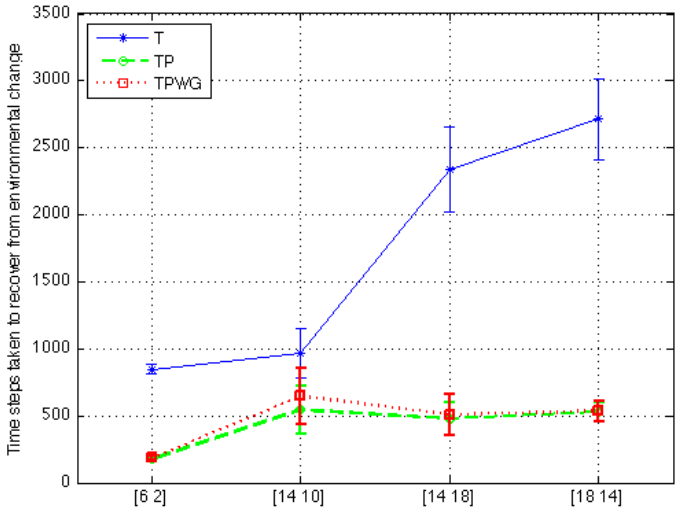

Figure 5.22: Time steps taken to recover from environmental changes after the time to reward changes from Poisson(6) and Poisson(10); the amount of reward for both arms (1) does not change.

In order to get the average performance of the algorithms, we also generate

the mean of the Poisson distributions, from which the time to reward for arms is drawn at every episode, randomly from an integer number discrete uniform distribution once at the first episode and once at the 1001th episode similar to what we have done in deterministic experiments. The amount of reward for both arms is 1 all the time.

1. Case 9: $t_1 \sim Poisson(\lambda_1)$, $\lambda_1 \sim IntegerDiscreteUniform(0, 100)$; $t_2 \sim Poisson(\lambda_2)$, $\lambda_2 \sim IntegerDiscreteUniform(0, 100)$; $t'_1 \sim Poisson(\lambda'_1)$, $\lambda'_1 \sim IntegerDiscreteUniform(0, 100)$; $t'_2 \sim Poisson(\lambda'_2)$, $\lambda'_2 \sim IntegerDiscreteUniform(0, 100)$
   The mean of the Poisson distributions used to generate the time to reward for the two arms is drawn both from an integer number discrete uniform distribution ranging from 0 to 100 once at the beginning and once when the environment changes. In this scenario, the mean of the time to reward for both arms may become shorter, longer or stay the same (even though the chance of which is very small). The rank of both arms on average may also stay the same or change when the environment changes.

2. Case 10: $t_1 \sim Poisson(\lambda_1)$, $\lambda_1 \sim IntegerDiscreteUniform(0, 50)$; $t_2 \sim Poisson(\lambda_2)$, $\lambda_2 \sim IntegerDiscreteUniform(51, 101)$; $t'_1 \sim Poisson(\lambda'_1)$, $\lambda'_1 \sim IntegerDiscreteUniform(0, 50)$; $t'_2 \sim Poisson(\lambda'_2)$, $\lambda'_2 \sim IntegerDiscreteUniform(0, 50)$
   The mean of the Poisson distribution used to generate the time to reward for the first arm is drawn from an integer number discrete uniform distribution ranging from 0 to 50 once at the beginning and once when the environment changes; the mean of the Poisson distribution used to generate the time to reward for the second arm is drawn from an integer number discrete uniform distribution ranging from 51 to 101 once at the beginning and from an integer number discrete uniform distribution ranging from 0 to 50 once when the environment changes. In this scenario, the mean of the time to reward for the first arm may become shorter, longer or stay the same (even though the chance of which is very small). The mean of the time to reward for the second arm certainly decreases. The rank of both arms on average changes from a certain rank to an uncertain rank when the environment changes.

3. Case 11: $t_1 \sim Poisson(\lambda_1)$, $\lambda_1 \sim IntegerDiscreteUniform(0, 50)$; $t_2 \sim$

$Poisson(\lambda_2)$, $\lambda_2 \sim IntegerDiscreteUniform(51, 101)$; $t'_1 \sim Poisson(\lambda'_1)$, $\lambda'_1 \sim IntegerDiscreteUniform(51, 101)$; $t'_2 \sim Poisson(\lambda'_2)$, $\lambda'_2 \sim IntegerDiscreteUniform(51, 101)$

The mean of the Poisson distribution used to generate the time to reward for the first arm is drawn from an integer number discrete uniform distribution ranging from 0 to 50 once at the beginning and from an integer number discrete uniform distribution ranging from 51 to 101 once when the environment changes; the mean of the Poisson distribution used to generate the time to reward for the second arm is drawn from an integer number discrete uniform distribution ranging from 51 to 101 once at the beginning and once when the environment changes. In this scenario, the mean of the time to reward for the first arm certainly increases and the mean of the time to reward for the second arm may become shorter, longer or stay the same (even though the chance of which is very small). The rank of both arms on average changes from a certain rank to an uncertain rank when the environment changes.

4. Case 12: $t_1 \sim Poisson(\lambda_1)$, $\lambda_1 \sim IntegerDiscreteUniform(0, 50)$; $t_2 \sim Poisson(\lambda_2)$, $\lambda_2 \sim IntegerDiscreteUniform(51, 101)$; $t'_1 \sim Poisson(\lambda'_1)$, $\lambda'_1 \sim IntegerDiscreteUniform(51, 101)$; $t'_2 \sim Poisson(\lambda'_2)$, $\lambda'_2 \sim IntegerDiscreteUniform(0, 50)$

   The mean of the Poisson distribution used to generate the time to reward for the first arm is drawn from an integer number discrete uniform distribution ranging from 0 to 50 once at the beginning and from an integer number discrete uniform distribution ranging from 51 to 101 once when the environment changes; the mean of the Poisson distribution used to generate the time to reward for the second arm is drawn from an integer number discrete uniform distribution ranging from 51 to 101 once at the beginning and from an integer number discrete uniform distribution ranging from 0 to 50 once when the environment changes. In this scenario, the mean of the time to reward for the first arm certainly increases and the mean of the time to reward for the second arm certainly decreases. The rank of both arms on average changes from a certain rank to another certain rank when the environment changes.

As figure 5.23 shows, similar to the conclusions for deterministic environments, TP/TPWG learning recovers from environmental changes faster than T learning

in all experimental scenarios. However, it is worth noting that the results of the three algorithms are very close in Case 9 and all are worse than those in deterministic environments. Similar to the results for deterministic environments, TP learning and TPWG learning perform almost the same in terms of the speed in recovering from environmental changes for similar reasons.

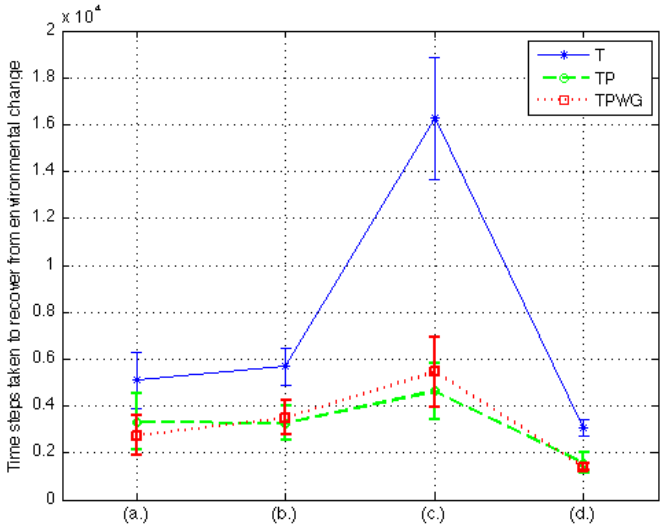

Figure 5.23: Time steps taken to recover from environmental changes after the time to reward changes; the amount of reward for both arms (1) does not change; (a.) Case 9; (b.) Case 10; (c.) Case 11; (c.) Case 12

We then use Case 9 as a test bed to experiment on T learning and TP/TPWG learning with different initial learning rates (α_0). The results are shown in 5.24. Unlike the counterpart experiment in deterministic environments, the time steps taken to recover from environmental changes by TP/TPWG learning do not monotonically decrease when the initial learning rate increases. This is because in stochastic environments a faster learning rate may cause the learning to become unstable.

5.5.3 When the amount of reward for actions may be different and may also change

We first compare the training process of the three algorithms. The following two cases are considered.

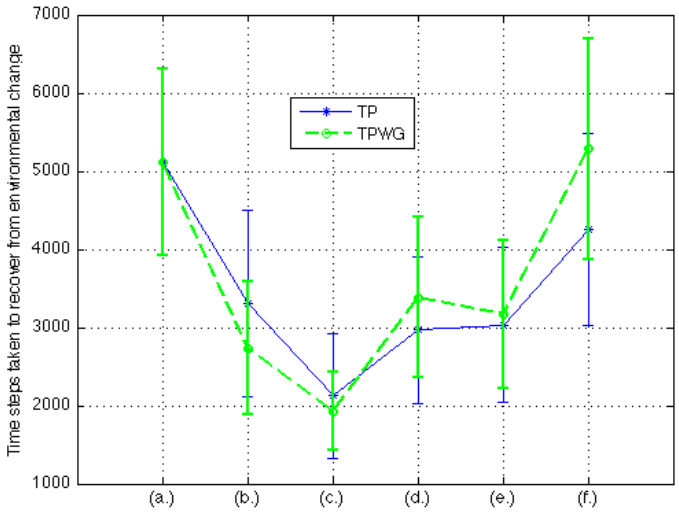

Figure 5.24: Time steps taken to recover from environmental changes with different initial learning rates; the mean of the Poisson distributions used to generate the time to reward for the two arms is drawn both from an integer number discrete uniform distribution ranging from 0 to 100 once at the beginning and once when the environment changes; the amount of reward for both arms (1) does not change.(a.) T learning ($\alpha = 0.1$); (b.) TP/TPWG learning with $\alpha_0 = 1.2$; (c.) TP/TPWG learning with $\alpha_0 = 1.6$; (d.) TP/TPWG learning with $\alpha_0 = 2.0$; (e.) TP/TPWG learning with $\alpha_0 = 2.4$; (f.) TP/TPWG learning with $\alpha_0 = 2.8$.

1. Case 1: $t_1 \sim Poisson(6), r_1 \sim Poisson(6), t_2 \sim Poisson(10), r_2 \sim Poisson(10)$
   The second action is optimal on average in terms of the discounted reward, but the difference between the two actions on average is relatively small.

2. Case 2: $t_1 \sim Poisson(6), r_1 \sim Poisson(100), t_2 \sim Poisson(10), r_2 \sim Poisson(10)$
   The first action is optimal on average in terms of the discounted reward, and the difference between the two actions on average is relatively big.

In Case 1, as figure 5.25 shows, VP/VPWG learning learns Q values faster than V learning due to a bigger learning rate when the estimated value is not correct, which is usually the case at the beginning of learning. Because the difference in Q values between the two actions is relatively small, giving up has not occurred and therefore there is little difference between the three algorithms after learning. When the difference in the Q value between the two actions increases, however, giving up takes effect. As figure 5.26 shows, VP learning performs better than V/VPWG learning after the learning converges in terms of the discounted reward received thanks to giving up when the difference in the Q value between actions is big. On average, the discounted reward obtained through VP learning in one episode after learning is a little more than, and also less noisy than, that obtained through V/VPWG learning. Compared with the deterministic case, the result is considerably noisier and the difference among the three algorithms is also less apparent.

Next, we compare their performance in terms of recovering from environmental changes. We experiment on the alorithms using the following two scenarios:

1. Case 3: $t_1 \sim Poisson(\lambda_1)$, $\lambda_1 \sim IntegerDiscreteUniform(0, 100)$;
   $t_2 \sim Poisson(\lambda_2)$, $\lambda_2 \sim IntegerDiscreteUniform(0, 100)$;
   $t_1' \sim Poisson(\lambda_1')$, $\lambda_1' \sim IntegerDiscreteUniform(0, 100)$;
   $t_2' \sim Poisson(\lambda_2')$, $\lambda_2' \sim IntegerDiscreteUniform(0, 100)$;
   $q_1 \sim Poisson(\lambda_3)$, $\lambda_3 \sim IntegerDiscreteUniform(0, 100)$;
   $q_2 \sim Poisson(\lambda_4)$, $\lambda_4 \sim IntegerDiscreteUniform(0, 100)$;
   $q_1' \sim Poisson(\lambda_3')$, $\lambda_3' \sim IntegerDiscreteUniform(0, 100)$;
   $q_2' \sim Poisson(\lambda_4')$, $\lambda_4' \sim IntegerDiscreteUniform(0, 100)$
   The means of the Poisson distributions used to generate the time to reward and the discounted reward for the two arms are independently drawn from an integer number discrete uniform distribution from 0 to 100 inclusive

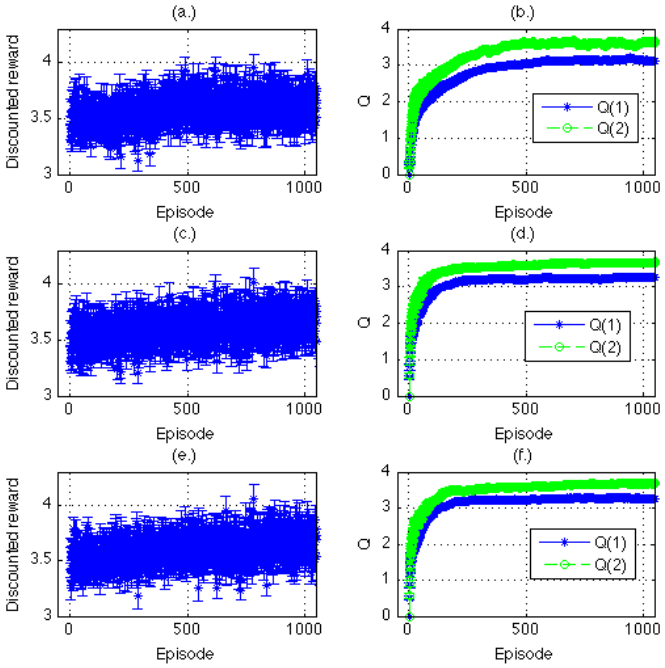

Figure 5.25: Comparison of the three algorithms during training; Case 1. (a.) discounted reward received by V learning; (b.) the learning of Q values by V learning; (c.) discounted reward received by VPWG learning; (d.) the learning of Q values by VPWG learning; (e.) discounted reward received by VP learning; (f.) the learning of Q values by VP learning.

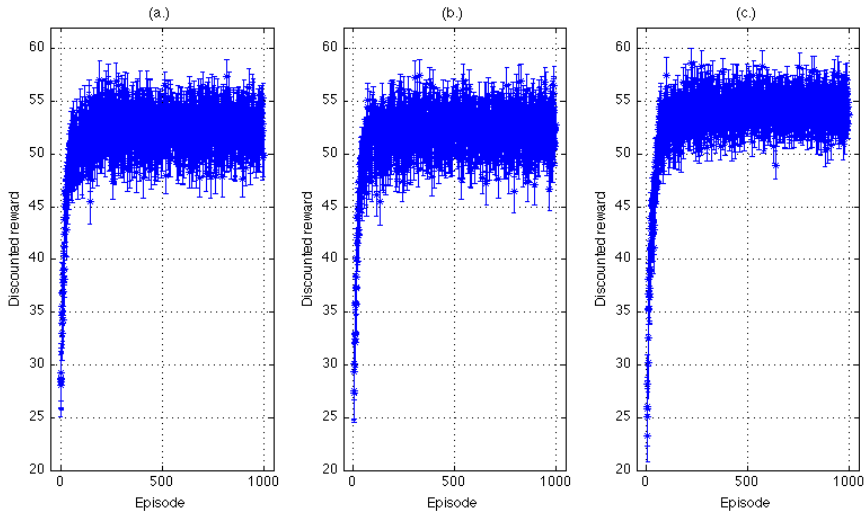

Figure 5.26: Comparison of the three algorithms during training; Case 2. (a.) discounted reward received V learning; (b.) discounted reward received VPWG learning; (c.) discounted reward received VP learning.

once at the beginning and once when the environment changes, and the amount of reward is deduced from them. The reason why we generate the discounted reward rather than the amount of reward for experiments is that the discounted reward decides the optimal action and we use the last change of the optimal action to measure the recovery time.

2. Case 4: $t_1 \sim Poisson(6), r_1 \sim Poisson(10), t_2 \sim Poisson(10)$, $r_2 \sim Poisson(100); mean(t'_1) = mean(t_1), r'_1 \sim Poisson(100)$, $mean(r'_1) > mean(r_1), mean(t'_2) = mean(t_2), mean(r'_2) = mean(r_2)$
The mean of the amount of reward for the first action increases and the first action becomes the optimal action on average in the new environment in place of the second one.

The first experiment tests their average performance. Similar to the results for the deterministic experiment, as figure 5.27 shows, both VP learning and VPWG learning recover from environmental changes more than twice as fast as V learning. It is worth noting that, similar to the results for the deterministic experiments, VPWG learning performs even better than VP learning for similar reasons. In some scenarios the amount of reward for the suboptimal action increases and the suboptimal action may become the optimal one. Due to giving up, it would take VP learning longer to find the change. To demonstrate this, we especially designed such a scenario, viz. Case 4. The result is shown in figure 5.28. VP learning with standard parameter settings performs worst of them. With the increase in ϵ_3 (the probability of exploring the amount of reward, viz. not giving up, increases), the performance of VP learning improves. VPWG learning performs best.

5.6 Conclusions and discussion

This chapter introduced the time delayed n-armed bandit problem, a simple problem with only one state but n actions, investigated possible implementations of learning and perceiving the time to reward, and designed simple algorithms specifically for this kind of reinforcement learning problems with only one state. The problem was then used as a test bed to compare the standard time/value

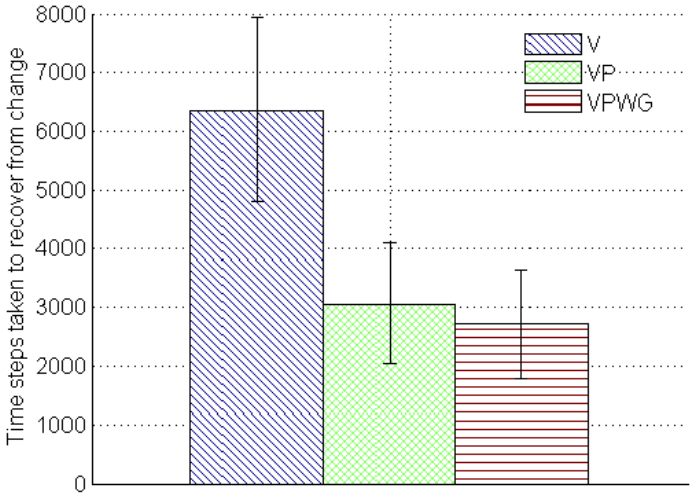

Figure 5.27: Time steps needed to recover from environmental changes after both the time to reward and the amount of reward change; Case 3.

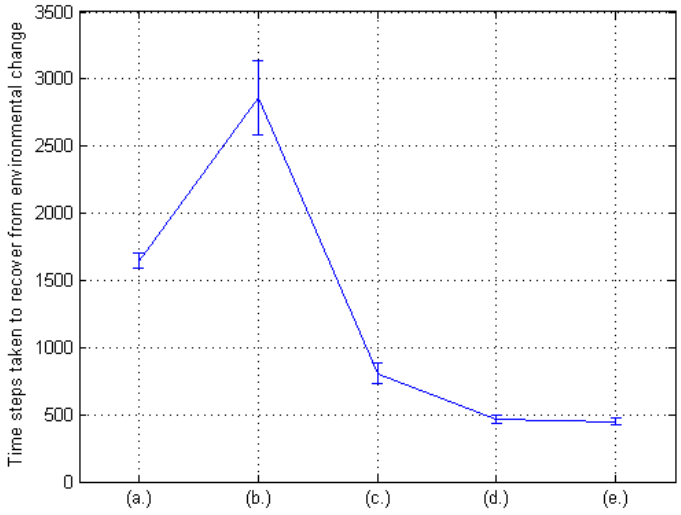

Figure 5.28: Time steps needed to recover from environmental changes; Case 4. (a.) V learning; (b.) VP learning, $\epsilon_3 = 0.1$; (c.) VP learning, $\epsilon_3 = 0.5$; (d.) VP learning, $\epsilon_3 = 1$; (e.) VPWG learning.

estimation reinforcement learning algorithms with time/value perception algorithms. We have considered both deterministic environments and stochastic environments. In both kinds of environments, we initially considered only the cases where the two arms have the same amount of reward but different time to reward, then the cases where the two arms have different amounts of reward and different time to reward. In all cases, we compared the performance of these learning algorithms in terms of the learning speed, the recovery speed from environmental changes, and the agent's performance after the learning converges. The experimental results show that the idea of time perception or value perception has significantly improved the performance of the learning agent in terms of the learning speed (for both TP/VP and TPWG/VPWG learning), the recovery speed from environmental changes (for both TP/VP and TPWG/VPWG learning) and the agent's performance after the learning converges (for TP/VP learning only) in most of the experimental cases.

In the following subsections, we will discuss what policy the algorithms introduced in this chapter learn, alternative models, and two modified problems, viz. when the reward may never come and when there is an energy/time limit.

5.6.1 On-policy or off-policy

For T/TPWG and V/VPWG learning, they are on-policy learners, viz. they use the same policy to update their estimation and make decisions (evaluate/improve the same policy while following it). For both TP and VP learning, however, they are not on-policy learners. For actions which are not given up, their time to reward or values are learned on-policy. For actions which are given up, their time to reward or values are learned off-policy. This is because their estimated time to reward or values are not updated when they are given up. In other words, the policy used to make decisions contains giving up whereas the policy used to update their estimation ignores giving up. It is worth pointing out, however, that both TP and VP learning can be modified to on-policy learners by updating the estimation of a given-up action with the actual time to reward spent in this action and other actions before a reward is received. For example, suppose the learning agent gives a_1 up t_1 time steps after a_1 is chosen. After giving up, suppose the learning agent chooses a_2 and then receives a reward t_2 after a_2 is chosen. In off-policy TP learning, the estimated time to reward for a_1 is not updated and only the estimated time to reward for a_2 is updated in this episode. In on-policy

TP learning, however, the estimated time to reward for a_1 is also updated with $t_1 + t_2$.

It is worth noting the following two points about on-policy TP or VP learning. Firstly, the estimated values of given-up actions are dependent on those of the non given-up actions through which a reward is received. For the previous example, the estimated time to reward for a_1 is dependent on the actual time to reward for a_2. If the time to reward for a_2 changes, the estimated time to reward for both a_1 and a_2 would become incorrect. In addition, we cannot demand that the time to reward for actions should be correctly learned before giving them up for on-policy TP or VP learning (unlike off-policy TP or VP learning). Otherwise, it may cause instability. For the previous example, suppose that the time to reward for a_1 has been correctly learned. When the learning agent gives up a_1, the estimated time to reward for a_1 is updated with $t_1 + t_2$ which may be different from the original time to reward without giving up and therefore it is very likely that the estimated time to reward for a_1 would become incorrect. If so, the learning agent is not allowed to give up a_1 and another cycle of switching between giving up and not giving up begins.

5.6.2 Alternative models

When the difference in the time to reward for actions is quite large, TP learning improves the performance of the learning agent by giving up the current action when it discovers that the action is still worse than the optimal action. Some may argue that the improvement can also be made by allowing the learning agent to make decisions whether or not to give up at every time step. This makes the problem more complex. In fact, this has transformed a one-state problem into a multiple-states problem as shown in figure 5.29 for a 2-armed bandit problem. The learning agent always starts in s_1. From s_1, if it chooses the first arm/action (a_1), it will enter s_2; on the other hand, if it chooses the second arm/action a_2, it will enter s_3. In both s_2 and s_3, it can choose to wait (a_1) and stays in the state, or can choose to give up (a_2), and then goes back to s_1 and remakes decisions.

When we use the standard Q learning on the new scenario where $t_1 = 6, t_2 = 10, r_1 = r_2 = 1$ and the parameters are as usual ($\alpha = 0.1, \epsilon = 0.1, \gamma = 0.9$), it does not work very well as shown in figure 5.30. Firstly, it takes a long time to get the first reward. This is because, whether in s_2 or s_3, the learning agent has an equal chance to give up and to wait at every time step initially thanks to the

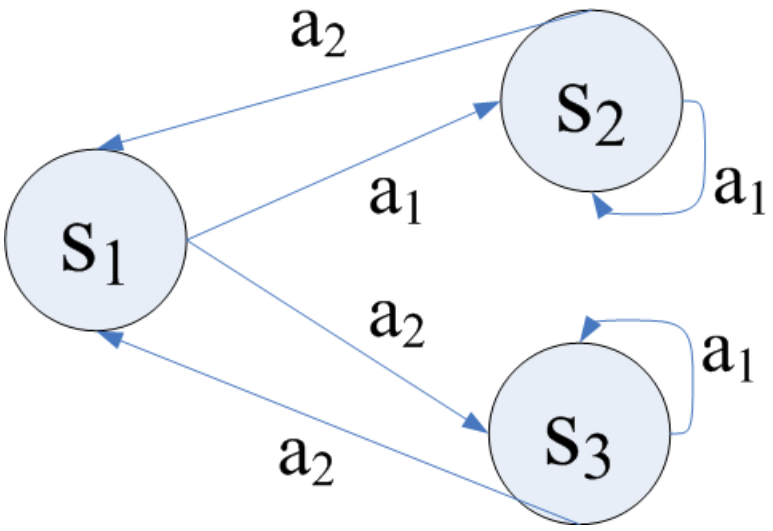

Figure 5.29: Modified time delayed n-arm bandit problem with options to give up at every time step; s: state; a: action; s_1: the starting state; s_2: the first arm; s_3: the second arm.

equal initial Q values of the two actions. If the learning agent chooses the first action in s_1 and enters s_2, it has only the probability 0.5^5 to receive a reward without giving up and has the probability $1 - 0.5^5$ to give up before receiving a reward. If the learning agent chooses the second action and enters s_3, it has even less probability to receive a reward without giving up. One way to solve this problem is to give the learning agent a small penalty when it gives up. This, however, would disadvantage giving up and make giving up less likely to happen even when it should happen afterwards. Another way is to set the initial Q value of the first action (wait) in both s_2 and s_3 bigger than that of the second action (give up). This, however, would disadvantage giving up and make it hard for the learning agent to learn the true Q value of the second action (give up). Secondly, the learning is unstable. This is because it cannot differentiate one state at a different time step. The Q values of one state at a different time step should be different. Take s_3 as an example, at the first time step, it may be better to give up to s_1 and then choose the first action; but at the 9th time step, it is better not to give up. It is worth pointing out that, when the time to reward for the two arms increases, both problems will become more serious.

One way to solve the problems is to augment the states with time steps. Previously, Q is associated with (s, a) pairs but now it is associated with (s, t, a)

triplets. The result is shown in figure 5.31. Firstly, it still shares the problem of taking a long time to get the first reward with the Q learning without time augmentation. Secondly, after learning, though it is much better than the Q learning without time augmentation, it is still worse than TP and even T/TPWG learning shown in figure 5.8. This is mainly due to non-greedy decision making. Take s_2 as an example, at every time step, the learning agent waits there with probability 0.95 (viz. $1-\frac{\epsilon}{2}$) after learning. But before the learning agent receives a reward, the probability of giving up is $1-0.95^5 = 0.23$. Therefore, the learning agent may give up several times before actually receiving a reward even when the optimal action is chosen in s_2 most of the time. Furthermore, it is not hard to predict that the result would become even worse if the time to reward for the two arms is longer because the probability of giving up increases when the time to reward increases. As mentioned previously, non-greedy decision making, however, is necessary in nonstationary environments. It is also worth noting that, when the learning agent chooses the second action in s_1 and enters s_3, the learning agent is more likely to give up at the first couple of time steps and less likely to give up at the last couple of time steps due to the rank of Q values of the two actions in s_3. This is a desirable behaviour in stationary environments. In nonstationary environments, however, this prevents the learning agent from detecting changes in the environment. For instance, when the time to reward for the second arm has decreased and become even less than that for the first arm, it would be difficult for the learning agent to detect, since the learning agent usually gives up the second arm before receiving the new reward for the second arm. In contrast, our algorithms always wait at least the estimated time to reward for the first arm if the second arm is chosen. Therefore, they can quickly detect the change if the second arm has become better than the first arm.

5.6.3 When the reward may never come

In the time delayed n-armed bandit problem discussed previously, the time to reward may become shorter or longer than previously, but the reward will eventually come. In some cases, however, the reward may never come or is so delayed that it is not worth waiting when the reward does not appear around the expected time. For example, in a foraging scenario, it may be due to the exhaustion of a foraging site (the environment has changed abruptly). Or, the destination of the learning agent is somewhere in the south, but it goes north. In these cases,

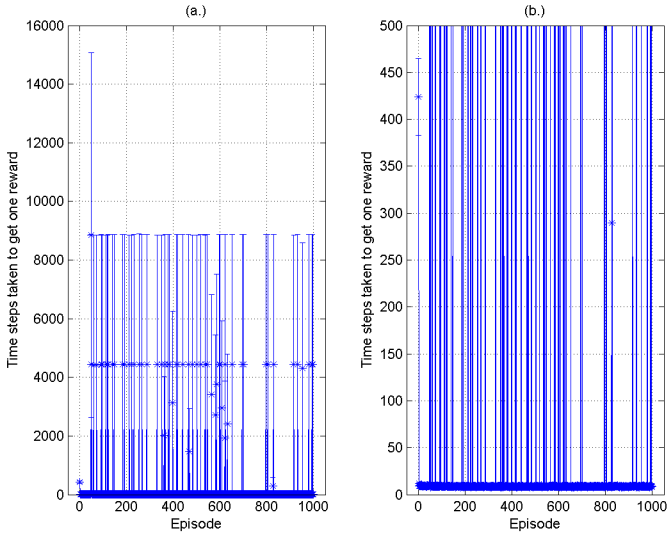

Figure 5.30: The learning process of Q learning during training in the new experimental setting where the learning agent is able to make a decision of whether or not to give up at every time step. (a.) the learning process; (b.) the zoomed-in version of (a.).

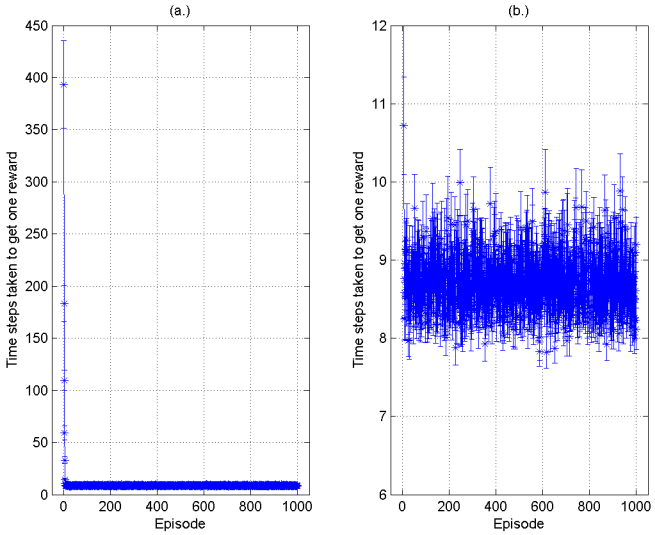

Figure 5.31: The learning process of Q learning with time augmentation during training in the new experimental setting where the learning agent is able to make a decision whether or not to give up at every time step. (a.) the learning process; (b.) the zoomed-in version of (a.).

the learning agent neither needs to compare with other actions nor needs to consider the amount of reward to decide when it should give up the current action. Suppose that the time to reward for a has been correctly learned. When the reward does not come within $T(a) + k\sqrt{T\_var(a)}$, the reward may never come and therefore the learning agent would give up a. It is worth noting, however, that the learning agent would never know whether or not the reward will appear in the near future unless it continues waiting and the 'future' arrives.

As mentioned previously, animal experiments also show that animals have the ability to learn the time to reward and would give up if the reward does not appear for some time. For instance, the experiment by Brunner et al. [26] on starlings showed that the rate of their pecking (active search of foods) peaked when the time was close to the time of the reward; the starlings stopped pecking about 1.5 times the inter-prey interval if they had not received any reward, and abandoned the current patch and flew to other patches 13s later.

5.6.4 When the energy budget is limited

In the previous experiments, there is no time/energy limit and the learning agent can wait after taking one action as long as it wishes. Unfortunately, however, this is not usually the case in the real-world. Firstly, every learning agent, whether biological or artificial, has a lifetime. The reward that comes after its lifetime means little, if not nothing, to it. Secondly, there are usually even tighter time/energy restrictions on them. For instance, predators cannot wait for prey without eating any for too long, or they will die from hunger. Likewise, a battery-powered robot cannot go without recharging for too long, or it will run out of battery. In this subsection, we consider cases where the learning agent has an energy budget at the beginning of every episode. After taking one action, its energy will elapse over time. We also assume that the learning agent knows how much energy it has at the beginning of every episode. This assumption is the usual case in the real-world, though the learning agent may have only rough/noisy knowledge of its energy instead of complete knowledge. Predators know how hungry they are and how long they can bear without eating any food. A battery-powered robot can also measure how much power is left in its battery.

The reward can be energy or an abstract value. If it is energy, a natural goal of the learning agent is to maximise the energy gained minus the energy lost. If it is an abstract value, a natural goal of the learning agent is to maximise the

discounted value obtained within the energy limit. In some real-world problems, the learning agent is required to come back to the starting point/ charging point before it runs out of energy. If the reward is an abstract value, it is quite straightforward. The learning agent can only use half of its energy limit to explore and has to leave another half to go back if the penalty for not being able to return outweighs the potential reward, which is normally the case. This is because the learning agent is definitely unable to return once it consumes more than half of its energy, assuming that it costs the learning agent the same amount of energy to go somewhere and come back. If the reward is energy, however, it becomes more complicated, The learning agent may be able to use all its energy budget for exploration and then uses the obtained reward energy to go back. But this is risky because the learning agent may not be able to get the reward energy before running out of energy. Furthermore, even if the learning agent can get the reward energy, the quantity of the reward energy may not be enough to support it on the return journey. The optimal policy depends on both the credit structure and the experimental settings. For simplicity, we only consider cases where the reward is an abstract value.

For the simplicity of analysis, we only consider the case of a 2-armed bandit problem here. We first consider cases where both the amount of and the time to reward are deterministic. The first arm has a reward of r_1 units occurred t_1 time steps after being pushed and the second one has a reward of r_2 units occurred t_2 time steps after being pushed. It is apparent that the learning agent should choose the first action if $r_1 > r_2$ and $t_1 \leq t_2$ and the second action if $r_1 < r_2$ and $t_1 \geq t_2$ regardless of the energy limit. Otherwise, the optimal policy may depend on the energy limit. In addition, if the energy is enough for the learning agent to get a reward for any action, the problem is equivalent to that without energy limit: the learning agent just chooses the optimal one. On the other hand, if the energy is not enough for the learning agent to get a reward in any action, there is no difference in choosing any of these actions, either. Furthermore, if the energy budget is enough for the optimal action, the problem also becomes similar to, though not exactly the same as, not having an energy limit: the learning agent just chooses the optimal one. Therefore, we only consider cases where the action, whose time to reward is shorter, has a smaller amount of reward and the energy budget is enough for the suboptimal action to get a reward but not enough for the optimal action. Even in this case, if

the energy budget is deterministic, the learning agent with the standard value estimation reinforcement learning algorithm, can still find the optimal action with the energy budget and therefore can behave optimally. This is the beauty of classical value estimation reinforcement learning algorithms.

For the above reasons, we consider a case which has a stochastic energy budget and satisfies the above conditions. Specifically, the experimental scenario is as follows. The first arm has a reward of 1 unit occurred 5 time steps after being pushed and the second one has a reward of 2 units occurred 10 time steps after being pushed. The energy of the learning agent is subject to a uniform distribution from 5 to 15 inclusive at the beginning of every episode and will decrease by 1 unit after each time step. Each episode ends either when a reward is received or when the agent's energy is exhausted. The experiment is run 100 times to reduce the influence of random noise. We experiment on three algorithms. The first algorithm is called V1 learning, a standard value estimation reinforcement learning algorithm which does not update its Q value when a reward is not received before the time limit. The second algorithm is called V2 learning, a standard value estimation reinforcement learning algorithm, which updates its Q value with 0 when a reward is not received before the time limit. The last algorithm is called VTP learning which combines V1 and TP/TPWG learning. It chooses the best possible action whose reward is predicted to be able to reach with the energy budget, viz. the estimated time to reward is less than the time/energy limit.

Suppose first that the learning agent has no time/energy limit at the beginning of learning (the first 1000 episodes) so that it can fully learn the environment. It is worth noting that we will remove this restriction later. The assumption seems not realistic at first glance. But it is also usually the case in the real world. When a child lion is under the training of its mother, it is not limited by its energy since it can get extra energy from its mother.

As figure 5.32 shows, V1 learning still considers the second arm as the optimal action and therefore chooses it most of the time. Without the time/energy limit, the second action is the optimal one. With the time/energy limit, however, the second action is even worse than the first one on average. V2 learning correctly identifies the first arm as the optimal action and therefore chooses it most of the time. VTP learning, however, chooses the second one when the learning agent has enough energy to reach its reward and the first one otherwise. Therefore, the learning agent with VTP learning can behave optimally whether it has enough

energy or not. From figure 5.32, we can also see VTP learning performs best among the three.

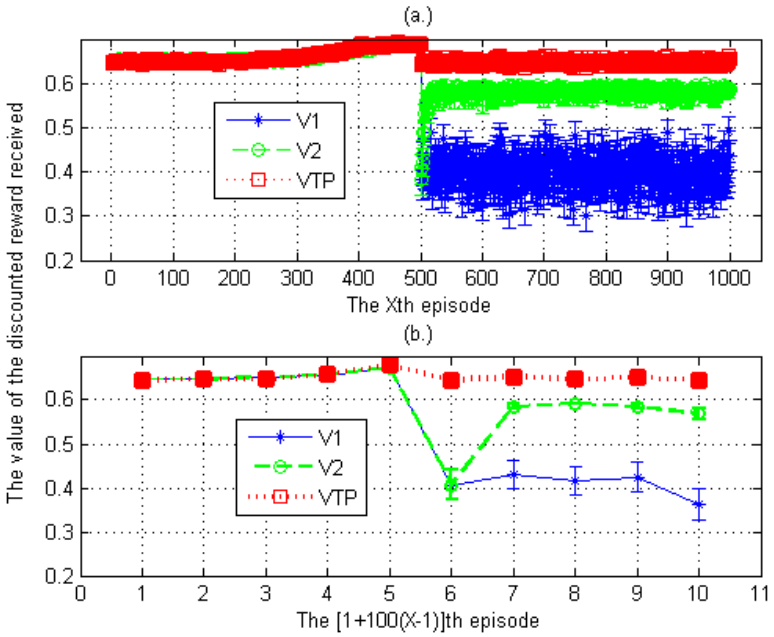

Figure 5.32: The comparison of the three algorithms in terms of the value of the discounted reward received. (a.) the value of the discounted reward received in the Xth episode; (b.) the value of the discounted reward received in the $[1+100(X-1)]$th episode. V1: a standard value estimation reinforcement learning algorithm which does not update its Q value when the reward is not received before the time limit; V2: a standard value estimation reinforcement learning algorithm which updates its Q value with 0 when the reward is not received before its energy is exhausted; VTP: Learn the time to reward which chooses the best possible action whose reward is predicted to be able to be reached with the energy.

The problem has become much easier with the above assumption that the learning agent has no time/energy limit at the beginning of learning. But it may not be true in some cases and therefore here we try to remove the assumption. There should be little difference for V1 and V2 learning because they do not learn the time to reward. For VTP learning which learns the time to reward, there may be some difficulties in learning the true time to reward for the second arm due to the energy limit. Therefore, we modify VTP learning in order that it first chooses the first arm exclusively, and only updates its Q value and the time to reward when a reward is received. When the time information for the first arm is correctly learned, it then chooses the second arm exclusively and learns its Q value

and the time to reward as stated above. It is worth noting that, when the reward does not come for many episodes in a row after a certain action is chosen, the reward for the action is marked unreachable and we also consider that the time information has been correctly learned. The process goes on until it learns the time to reward for all arms correctly. Then, it will choose the optimal one when the learning agent has enough energy to reach its reward, otherwise the second optimal one when the learning agent has enough energy to reach its reward, and so on. In case the environment changes, it should also explore suboptimal actions with a small probability, if the actions have no chance of being taken otherwise.

We do experiments on the three algorithms in the same scenario but with the assumption removed. As figure 5.33 shows, the result is similar to the above result as shown in figure 5.32.

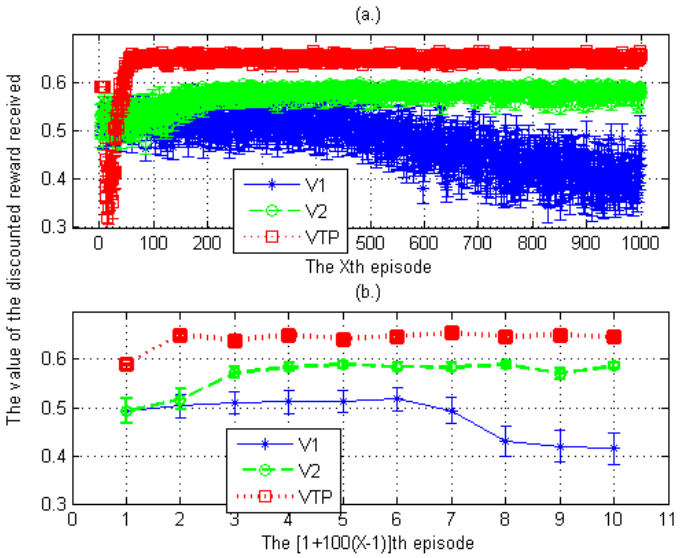

Figure 5.33: The comparison of the three algorithms in terms of the value of the discounted reward received. V1: a standard value estimation reinforcement learning algorithm which does not update its Q value when the reward is not received before the time limit; V2: a standard value estimation reinforcement learning algorithm which updates its Q value with 0 when the reward is not received before its energy is exhausted; VTP: Learn the time to reward which chooses the best possible action whose reward is predicted to be able to be reached with the energy.

If the amount of and the time to reward are stochastic but the time/energy limit is deterministic, V2 learning, which updates its Q value with 0 when the reward is not received before its energy is exhausted, is still optimal. When

the time limit/ energy budget is also stochastic, however, the problem becomes more complex. The optimal policy may depend on the distributions of both the time/energy limit and the time to reward. As a simple example, assume that r_1, r_2 and t_1 are deterministic, $r_1 < r_2$, and only t_2 is stochastic. Also assume that the optimal target is to maximise the undiscounted rewards. Then the learning agent needs to learn the $\frac{r_1}{r_2}$ percentile point of t_2 denoted as t_{2p}. If the learning agent has the time limit more than t_{2p} or less than t_1, it chooses the second action; otherwise, it chooses the first action. This policy is optimal if r_1, r_2, t_1 and t_{2p} have been learned correctly.

It is also possible that after the learning agent gets a reward from one arm, the learning agent still has enough energy left to try another episode or even more. In essence, one episode in this scenario is equivalent to a concatenation of several episodes in the above scenario. So, we will not consider this scenario specifically.

In this subsection, we have shown that learning the time to reward can improve the performance of the learning agent when the energy/time budget is limited. Animal experiments [18] also show that animals make different decisions on where to forage depending on their energy budget at that time.

Chapter 6

Route finder problem

In this chapter, the route finder problem is first introduced. The algorithms introduced in the last chapter are then extended to work with multiple states. Next, the settings of experiments are discussed and the results of experiments on these algorithms are presented. The last section concludes this chapter and discusses related questions.

6.1 Introduction

In the last chapter, we studied the time delayed n-armed bandit problem which has only one state. In this chapter, we study a problem with multiple states, viz. the route finder problem, which naturally extends the time delayed n-armed bandit problem. In the time delayed n-armed bandit problem, the learning agent makes a decision, takes the decision/action and then waits for a reward. In the route finder problem, however, after making one decision the learning agent may need to make another or even more decisions before receiving a reward.

As figure 6.1 illustrates, at every junction, the learning agent makes a decision on which path to choose, just like which arm to choose in the time delayed n-armed bandit problem. The length of each path varies similar to the time to reward for each arm in the time delayed n-armed bandit problem, so it can also be modelled as a semi-MDP by only modelling junctions as states in order to speed up learning and planning. In addition, similar to the time delayed n-armed bandit problem, we also treat the temporally extended actions or state transitions as temporal abstractions of an underlying MDP. Therefore, the agent has the option to change the course of the temporally extended actions or state

transitions, in particular, to give up the current route at every time step before reaching its destination, go back to any junctions that it has previous visited, and then remake its choice. There is, however, one difference from the time delayed n-armed bandit problem when the agent gives up the current action: the agent has to consider the extra cost caused by giving up because it needs to walk back to the junction to which it gives up. In other words, there is a cost to give up the current action and also a cost to follow it. Both must be considered in order to decide whether or not it is worthwhile to give up.

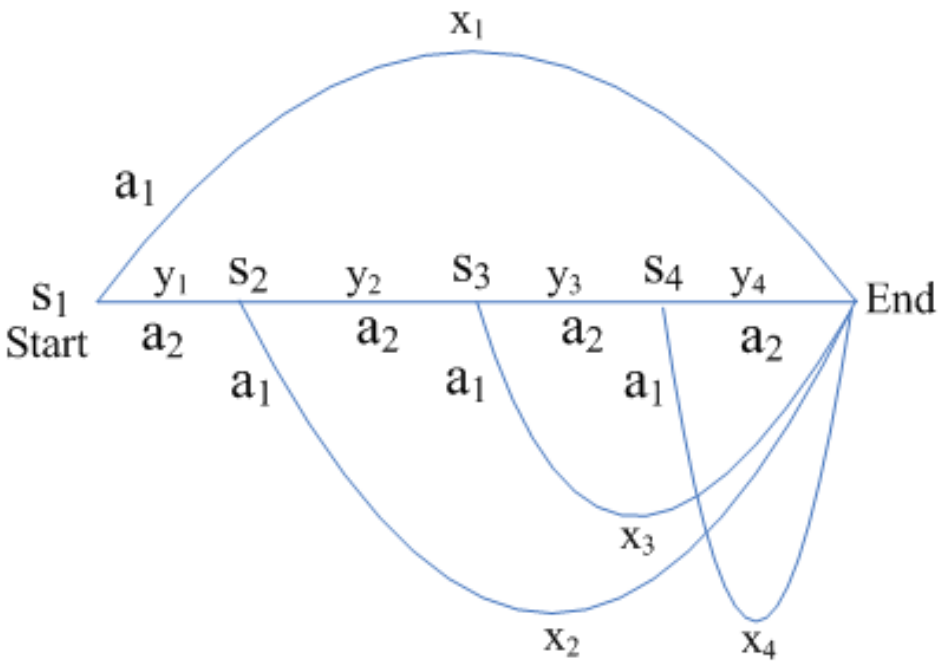

Figure 6.1: Illustration of a route finder problem

In fact, we can consider that the route finder problem is composed by a sequence of time delayed n-armed bandit problems. At every junction, the learning agent makes a decision on one time delayed n-armed bandit problem.

In this problem, only junction states, denoted as J, are high abstraction of states and have options. Since a normal action can also be considered as a special case of an option because an option lasts one or more time steps whereas a normal action lasts exactly one time step, we only consider options in these junction states. An option, denoted as o, begins from one junction state and ends at a neighbouring junction state or the destination state. Non junction states except the destination state, denoted as N, are considered as part of options.

In each junction state, denoted as s^J ($s^J \in J$), the agent chooses one from all options available in s^J based on its normal policy $\pi : J, O \rightarrow [0, 1]$ where O is a set of all options. After the agent selects one option, it follows another policy $\pi_o : N, A \rightarrow [0, 1]$ where A is a set of all actions available in the states N (viz. follows or gives up the current path), until the option terminates, viz. when the agent reaches one junction state again or the destination state. If it is not the destination state, the agent chooses one from all options available in the state based on its normal policy π similar to the above and another iteration begins. On the other hand, if it is the destination state, the agent receives a reward, denoted as r, and the current episode ends.

Suppose that the agent is in a junction state $s^J \in J$ (viz. outside options) at the i^{th} time step. Then, the value of taking option o in s^J under policy π in terms of the discounted return can be defined as

$$Q^{\pi}(s^J, o) = E(\sum_{k=i+1}^{n} \gamma^{k-i-1} r_k) \tag{6.1}$$

where n is the time step when the agent arrives at its destination, γ ($0 \leq \gamma \leq 1$) is a parameter called the *discount factor*, and r_k is the reward received at the k^{th} time step. In this problem, only when the agent reaches its destination will it receive a reward r. Therefore, the above equation can be further simplified as

$$Q^{\pi}(s^J, o) = E(\gamma^{n-i-1} r). \tag{6.2}$$

On the other hand, if the agent is in a non junction state $s^N \in N$ (viz. inside an option) at the i^{th} time step, the value function for s^N under policy π_o in terms of the discounted return can be defined as

$$V^{\pi_o}(s^N) = E(\sum_{k=i+1}^{n} \gamma^{k-i-1} r_k) \tag{6.3}$$

Similarly, the above equation can also be further simplified as

$$V^{\pi_o}(s^N) = E(\gamma^{n-i-1} r). \tag{6.4}$$

Instead of its maximising the discounted return, another natural goal of the agent is to walk from the starting point to the end point in the shortest time, viz.

to minimise the expected time to reward, which is used in this thesis.

Suppose that the agent is in a junction state $s^J \in J$ (viz. outside options) at the i^{th} time step and it takes the agent $t_{s^J,o}$ to reach its destination after an option o is taken in the state. Then, the value of taking option o in s^J under policy π with this criterion can be defined as

$$Q^{\pi}(s^J, o) = E(t_{s^J,o}). \tag{6.5}$$

On the other hand, if the agent is in a non junction state $s^N \in N$ (viz. inside an option) at the i^{th} time step and it takes the agent t_{s^N} to reach its destination from the state, the value function for s^N under policy π_o with this criterion can be defined as

$$V^{\pi_o}(s^N) = E(t_{s^N}) \tag{6.6}$$

At first glance, this problem seems like the single-source shortest-path problem [169] in a weighted graph with nonnegative weights, which can effectively be solved by Dijkstra's algorithm [170]. But the algorithm assumes that all vertexes and routes are known, and their weights/lengths are deterministic and stationary. In the problem we discuss here, however, the learning agent does not know the routes in advance and needs to explore them first. Furthermore, the length of paths may be stochastic and may also change over time. Therefore, a reinforcement learning algorithm is suitable for this problem.

Similar to the time delayed n-armed bandit problem, when the environment changes, it may be better to increase the learning rate in order to learn the change quickly and to increase the exploration rate (e.g. ϵ for ϵ-greedy) in order to increase the chance that suboptimal actions are visited if any suboptimal action has improved and can potentially become the optimal action in the new environment. When the environment does not change, on the other hand, it may be better to decrease the learning rate in order for the estimated value to converge to its true value and to decrease the exploration rate in order to reduce the cost of exploring suboptimal actions. Likewise, when the lengths of routes vary greatly, the learning agent may benefit from giving up the current route in order to avoid longer than necessary exploration if it has already found that the current route is still worse than the optimal one. However, the analysis becomes much more complex because this problem has more than one state. Even if the length of every route is deterministic, the estimated value of every state-action

pair is stochastic because the learning agent may make different decisions at the same junction due to the non-greedy decision making. Furthermore, some states far away from the starting point are seldom visited and their estimated values may be incorrect all the time. Using this more complex problem, we would like to find out if the idea of learning and perceiving the time to reward, which works very well in the relatively simple time delayed n-armed bandit problem, can still work well in more complex problems.

Here, we only consider cases where there is only one destination (reward). In addition, we further assume that the walking speed of the learning agent is the same everywhere, so we can still learn the time to reward instead of the length of routes. With these conditions, T learning and TP/TPWG learning introduced in the last chapter, if extended to multiple states, are sufficient to solve this problem.

6.2 Algorithms

In this section, we introduce a Monte Carlo method to learn the time to reward and another Monte Carlo method to learn and perceive the time to reward. The notation used to describe algorithms is summarised in table 6.1.

From the experimental results of the last chapter, we know that there is little difference between TP and TPWG learning or between VP and VPWG learning in terms of the learning speed and the recovery speed from environmental changes. TP or VP performs better than TPWG/VPWG learning in terms of the agent's performance after the learning converges only when the difference in the time to reward for actions or in the values of actions is great. Thus, we only extend TP learning to work with multiple states in this chapter.

Table 6.1: Summary of notation used to describe algorithms for the route finder problem

Notation	Meaning
S	set of possible states
s	any state $\in S$
s_c	the current state

s_1	the initial state
s_g	the state to which it should give up; 0 (no state) if it should not give up
$A(s)$	set of possible actions in state s
a	any action in one state
$A(s)\backslash a$	set of actions except action a in the state s
a_c	the current action
$a^*(s)$	the optimal action in s in terms of the agent's criterion of optimality
a_g	the action the agent will choose after giving up
α	learning rate, $0 < \alpha \leq 1$, a scalar with fixed value for MCT learning and a function of state-action pairs for MCTP learning
α_0, δ, η	parameters used to calculate α for MCTP learning
$\alpha_2(s, a)$	learning rate specifically used to learn the variance of the time to reward for (s, a), $0 < \alpha_2 \leq 1$
α_{2max}, α_{2min}	the maximum/minimum of α_2
ϵ	random parameter to explore suboptimal actions, $0 \leq \epsilon \leq 1$, a scalar with a fixed value for MCT learning and a function of states for MCTP learning
ϵ_{max}, ϵ_{min}	the maximum/minimum of ϵ
ϕ	parameter used to calculate ϵ for MCTP learning
ϵ_2	random parameter to decide whether to explore the current action longer than usual; with probability $\frac{\epsilon_2}{2}$, explore the current action longer than usual, $0 \leq \epsilon_2 \leq 1$
trackList	a list storing all state-action pairs visited in the current episode
$t(s, a)$	discrete time step elapsed since state-action pair (s, a) is visited

$T(s,a)$	estimated mean of the time to reward for state-action pair (s,a)
$T\_var(s,a)$	estimated variance of the time to reward for state-action pair (s,a)
k	the size of the expectation window ranging from $T(s,a)-k\sqrt{T\_var(s,a)}$ to $T(s,a)+k\sqrt{T\_var(s,a)}$
$count\_correct(s,a)$	the number of times/episodes in a row that the actual time to reward for state-action pair (s,a) is correctly estimated; whether or not they are correctly estimated is judged by algorithms.
$threshold(s,a)$	the time step after which the agent should give up (s,a)
$action\_forCmp(s,a)$	the action which the agent uses to compare with a in s; 0 (viz. no action) if no action is eligible
$flag\_correctEst(s,a)$	a Boolean storing the information about whether or not the time to reward for state-action pair (s,a) is correctly estimated judged by algorithms
$flag\_correctCmp(s,a)$	a Boolean storing the information about whether or not a can be used to compare with other actions in s judged by algorithms

6.2.1 Monte Carlo methods

The time estimation algorithm introduced in the last chapter is designed to work with only one state. The Monte Carlo method to be introduced in this subsection is a natural extension of the algorithm in order to work with multiple states. Like the time estimation algorithm, it also learns the estimated time to reward from experiences or samples and then updates the estimated time only at the end of an episode. Furthermore, the estimated time is also used to make decisions. But unlike the time estimation algorithm, the estimated time to reward in the Monte Carlo method is associated with state-action pairs rather than just actions because there may be more than one state in every episode. For the same reason, a track list is used to record all state-action pairs visited during one episode and a timer for each state-action pair in the track list is used to record how far/long the learning agent has gone since the visit to that particular state-action pair. After an episode ends, the timer information is used to update the estimated time to

reward for all of the state-action pairs in the track list incrementally.

The final algorithm of the Monte Carlo method for this problem is shown in algorithm 9. Please refer to table 6.1 for the meaning of notation used in the algorithm. Originally, the track list is empty. When a junction is encountered, the learning agent uses a non-greedy policy to choose one action from all actions available in this state with respect to their estimated time to reward. Then the state and the chosen action pair is added to the track list if it has not yet been in the list. Next, the learning agent moves forward one step along the chosen action/path and the timer of all state-action pairs in the track list is increased by one time step. The process will repeat until the learning agent arrives at the destination, signalling the end of an episode. At the end of one episode, the estimated time to reward for all state-action pairs in the track list is updated with their timer information (their actual time to reward) incrementally. It is worth noting that this algorithm is not our original contribution but a standard Monte Carlo algorithm to learn the time to reward.

Algorithm 9 A Monte Carlo method to learn the time to reward (MCT)

Inputs: T, s_1; **Outputs:** T; **Parameters:** α, ϵ; **Internal variables:** t, $trackList$
for all episode **do**
  $s_c = s_1$, $trackList = []$, $t = 0$
  repeat
    if the learning agent arrives at a junction (s_c) **then**
      Choose a_c from all possible actions in s using a non-greedy policy derived from $T(s_c, :)$ (e.g., ϵ-greedy, minimum T priority); then take action a_c
      if (s_c, a_c) is not already in $trackList$ **then**
        add (s_c, a_c) to $trackList$
      end if
    end if
    The learning agent moves forward one step
    For all (s, a) in $trackList$, $t(s, a) \leftarrow t(s, a) + 1$
  until the learning agent arrives at its destination
  For all (s, a) in $trackList$, $T(s, a) \leftarrow T(s, a) + \alpha \{t(s, a) - T(s, a)\}$
end for

6.2.2 Monte Carlo methods with time perception

Just like algorithm 9 extending the time estimation algorithm, the algorithm to be discussed in this section extends the algorithm of learning and perceiving the

time to reward in last chapter in order to work with multiple states. As shown in algorithm 10, it uses Monte Carlo methods to learn both the mean and variance of the time to reward, and then uses the learned time information to detect environmental changes and decide when to give up the current state-action pair. As shown in algorithm 11, if an environmental change is detected twice in a row, the learning rate is increased in order to learn the change quickly and the exploration rate (e.g. ϵ for ϵ-greedy) is increased in order to increase the chance that suboptimal actions are visited if one suboptimal action has improved and can potentially become the optimal action in the new environment; otherwise, the learning rate is decreased gradually towards 0 in order for the estimated mean of the time to reward to converge to its true mean and the exploration rate is also decreased gradually towards its minimum value in order to reduce the cost of exploring suboptimal actions. As shown in algorithm 12, if the learning agent has found that the current action is still worse than the optimal one, it gives up the action in order to avoid longer than necessary exploration.

Algorithm 10 is similar to algorithm 9 except that it also learns the variance of the time to reward and then uses the time information (both the mean and variance) to detect environmental changes and to decide when to give up the current action in order to avoid longer than necessary exploration. Algorithm 11 and algorithm 12 are also very similar to their counterpart algorithms 3 and 4 in the last chapter except that they augment all parameters and values with states since this problem has multiple states.

Specifically, when the actual time to reward $t(s,a)$ of the state-action pair (s,a) is outside $T(s,a) \pm k\sqrt{T\_var(s,a)}$ where $T(s,a)$ and $T\_var(s,a)$ are the sample mean and variance of the time to reward for the state-action pair (s,a), we consider that the time to reward for the state-action pair (s,a) has changed. If this happens twice in a row, a bigger learning rate is used to update $T(s,a)$ in order to learn the change quickly. Otherwise, the learning rate is decreased gradually towards 0 in order for the estimated value to converge to the true value. When the actual time to reward $t(s,a)$ of the state-action pair (s,a) is less than $T(s,a) - k\sqrt{T\_var(s,a)}$, we consider that the time to reward for the state-action pair (s,a) has become shorter. If so and also a is one suboptimal action in s and $t(s,a) < T(s,a^*)$, which means that a may have become the optimal action in s in the new environment, $\epsilon(s)$ will be increased so that a can be explored more often. Otherwise, the exploration rate is decreased gradually towards its minimum value

in order to reduce the cost of exploring suboptimal actions.

At every time step, the learning agent also checks whether it should give up the current action in order to avoid longer than necessary exploration. It is worth pointing out that if the time to reward for all state-action pairs has been correctly learned, the learning agent can not only detect when it should give up the current action but also find to which state it should give up by comparing the elapsed time since the visit to one action in one state with its estimated time to reward and with the estimated time to reward for other actions in the state. It checks all state-action pairs in the track list from the last to the first. One reason why it checks from the last to the first is that the cost of giving up to a latter visited state is smaller than giving up to an earlier visited state. Another reason is that the chance of giving up to a latter visited state is usually greater than that of giving up to an earlier visited state. Normally, the closer to the reward the less there is variation in the values of the state-action pair. For each state-action pair (s, a), it first checks if there exists any action $(a' \in A(s) \backslash a)$ in the state (s) that has the correct estimated time to reward for comparison (viz. $flag\_correctCmp(s, a') = TRUE$). If there is none, it will not give up the current action. On the other hand, if there is any, it finds the action (a'') which has the shortest estimated time to reward among all these actions. It will not give up unless the estimated time to reward for the state-action pair (s, a) is more than triple the estimated time to reward for the compared state-action pair (s, a''). If this condition is satisfied, most of the time, the learning agent only explores (s, a) for $T(s, a'')$ when (s, a) is chosen; But at times, it explores the pair for $3T(s, a'')$ just in case the time to reward for (s, a) has become less than $3T(s, a'')$. When $T(s, a) < 3T(s, a'')$, the learning agent is expected to get a reward in less than $3T(s, a'')$ if it does not give up and in $3T(s, a'')$ if it gives up after a is chosen in s. Therefore, it is better not to give up in this case.

6.3 Experimental settings

Each experiment, except the experiments with random data, is run 100 times to reduce the influence of random noise. When we experiment on the algorithms with random data, we generate 100 random cases and each case is only run once. In addition, these cases are generated in advance and then used for all algorithms to make the experimental comparison fair.

Algorithm 10 A Monte Carlo method to learn and perceive the time to reward (MCTP)

Inputs: T, T_var, s_1; **Outputs:** T, T_var
Parameters: k, ϵ_{min}; **Internal variables:** s_c, a_c, a_g, t, $trackList$, ϵ (initial value: ϵ_{min}), $count\_correct$ (initial value: 0), $threshold$ (initial value: ∞), $action\_forCmp$ (initial value: 0), $flag\_correctEst$ (initial value: $FALSE$), $flag\_correctCmp$ (initial value: $FALSE$)
for all episode **do**
  Initialise $trackList = []$, $t(s,a) = 0$ for all (s,a) pairs, $a_g = 0$, $s_c = s_1$
  repeat
    if the learning agent arrives at a junction (s_c) **then**
      if $a_g \neq 0$ **then**
        $a_c = a_g$, $a_g = 0$
      else
        Choose a_c from all possible actions in s_c using a non-greedy policy derived from $T(s_c, :)$ (ϵ-greedy, minimum T priority)
      end if
      if (s_c, a_c) is not already in $trackList$ **then**
        add (s_c, a_c) to $trackList$,
        use algorithm 12 to calculate $threshold(s_c, a_c)$
      end if
    end if
    For all (s, a) in $trackList$, $t(s,a) \leftarrow t(s,a) + 1$
    for all (s, a) in trackList **do**
      if $t(s,a) > T(s,a) + k\sqrt{T\_var(s,a)}$ **then**
        $flag\_correctEst(s,a) = flag\_correctCmp(s,a) = FALSE$
        Use algorithm 12 to recalculate the $threshold$ of the actions ($a' \in A(s) \backslash a$) where $action\_forCmp(s,a') = a$
      end if
    end for
    for all (s, a) in $trackList$ from the end to the beginning **do**
      if $flag\_correctEst(s,a) = TRUE\ AND\ t(s,a) > threshold(s,a)$ **then**
        $s_c = s, a_g = action\_forCmp(s,a)$ {Give up to s and then choose a_g}

        Reset t of the state-action pairs in $trackList$ ranging from (s,a) to the last pair, and remove these pairs from $trackList$; for state-action pairs in $trackList$ before (s,a), increase their timer by $t(s,a)$
        break
      end if
    end for
  until the learning agent arrives at its destination
  Use algorithm 11 to update the model
end for

Algorithm 11 Update the model used by algorithm 10

Inputs: T, T_var, t, ϵ, $count\_correct$, $trackList$
Outputs: $flag\_correctEst$, $flag\_correctCmp$, T, T_var, ϵ, $count\_correct$
Parameters: α_0, $\alpha_{2max}, \alpha_{2min}$, $\epsilon_{max}, \epsilon_{min}$, ϕ, k, δ, η; **Internal variables:** α, α_2, $a^*(s)$, T_{old}
for all (s, a) in $trackList$ **do**

$$flag\_correctCmp(s,a) = \begin{cases} TRUE & t(s,a) \leq T(s,a) + k\sqrt{T\_var(s,a)} \\ FALSE & \text{otherwise} \end{cases}$$

if $T(s,a) - k\sqrt{T\_var(s,a)} \leq t(s,a) \leq T(s,a) + k\sqrt{T\_var(s,a)}$ **then**
$flag\_correctEst(s,a) = TRUE$, $\alpha_2(s,a) = \alpha_{2max}$,
$count\_correct(s,a) \leftarrow count\_correct(s,a) + 1$
else
$flag\_correctEst(s,a) = FALSE$, $\alpha_2(s,a) = \alpha_{2min}$
if this happens twice in a row **then**
$count\_correct(s,a) = 0$
end if
end if
$\alpha(s,a) = \frac{\alpha_0}{(count\_correct(s,a)+1+\delta)^\eta}$
if a is not the optimal action **then**
if $t(s,a) < T(s,a) - k\sqrt{T\_var(s,a)}$ AND $t(s,a) < T(s,a^*)$ **then**
$\epsilon(s) = \epsilon(s) + \phi\,[\epsilon_{max} - \epsilon(s)]$
else
$\epsilon(s) = \epsilon(s) + \phi\,[\epsilon_{min} - \epsilon(s)]$
end if
end if
{Update the estimation}
$T_{old}(s,a) = T(s,a)$; $T(s,a) \leftarrow T(s,a) + \alpha(s,a)\,[t(s,a) - T(s,a)]$
$T\_var(s,a) \leftarrow T\_var(s,a) + \alpha_2(s,a)\{[t(s,a) - T_{old}(s,a)][t(s,a) - T(s,a)] - T\_var(s,a)\}$
end for

Algorithm 12 Calculate when it should give up used by algorithm 10

Inputs: T, $flag\_correctCmp$, (s, a), $A(s)$
Outputs: $threshold$, $action\_forCmp$
Parameters: ϵ_2; **Internal variables:** a', A', a''
if $\exists a' \in A(s) \backslash a$ satisfying $flag\_correctCmp(s, a') = TRUE$ **then**
  use A' to represent the set of all a', $a'' = \arg\min_{a' \in A'} [T(s, a')]$
  $action\_forCmp(s, a) = a''$
  if $T(s, a) \leq 3T(s, a'')$ **then**
    $threshold(s, a) = \infty$
  else
    $threshold(s, a) = \begin{cases} T(a'') & \text{with probability } 1 - \epsilon_2/2 \\ 3T(a'') & \text{with probability } \epsilon_2/2 \end{cases}$
  end if
else
  $action\_forCmp(s, a) = 0, threshold(s, a) = \infty$
end if

Like previous experiments, we consider that the learning has converged if the behaviour of the learning agent does not change afterwards and we consider that the learning has recovered from environmental changes if the learning agent can behave correctly in the new environment. However, there are two things worth considering. Firstly, because there is more than one state in this problem, some state-action pairs are seldom visited in some cases. For instance, as figure 6.1 illustrates, if the first action in all states is the optimal action, in one episode, the chance that the second action in s_1 is taken is $\epsilon/2$, the chance that the second action in s_2 is taken is $\epsilon^2/4$, the chance that the second action in s_3 is taken is $\epsilon^3/8$ and so on. It would take a long time if we require the behaviour of the learning agent in every state to be correct. Secondly, the optimal behaviour of the learning agent also depends on its policy. In this problem, for instance, the optimal path for a greedy learner is just the shortest path whereas the optimal path for a non-greedy learner is not necessarily the shortest path. For example, in the cliff-walking task [2], the optimal behaviour for a non-greedy learner is not to follow the shortest path which is close to the cliff but to follow a longer path which is far away from the cliff. For the same reason, the optimal path for a non-greedy learner with giving up may also be different. Therefore, we consider that the learning has recovered from environmental changes if the optimal route decided by its estimated values is the correct one with respect to its policy and does not change afterwards.

Notation	Meaning
$x_1, x_2, x_3, x_4, y_1, y_2, y_3, y_4$	the length of paths shown in figure 6.1 before the environment changes
$x'_1, x'_2, x'_3, x'_4, y'_1, y'_2, y'_3, y'_4$	the length of the above paths after the environment changes

Table 6.2: Summary of notation used to describe experimental scenarios for the route finder problem

In order to make the experiments tractable, we use a route finder problem with 4 junctions and 2 actions in each junction as figure 6.1 shows. The settings of parameters used in the following experiments are as follows where applicable. $\alpha = 0.1, \alpha_0 = 1.2, \alpha_{2min} = 0.002, \alpha_{2max} = 0.01, \epsilon = 0.1, \epsilon_{min} = 0.1, \epsilon_{max} = 1, \epsilon_2 = 0.1, \phi = 0.1, \gamma = 0.9, k = 3, \delta = 3, \eta = 1$. Initialise $T(s,a) = 0$, $T\_var(s,a) = 0$, $action\_forCmp(s,a) = 0$, $threshold(s,a) = \infty$, $flag\_correctEst(s,a) = FALSE$, $flag\_correctCmp(s,a) = FALSE$, $count\_correct(s,a) = 0$ for all (s,a) pairs.

6.4 Experimental results

In this section, we compare the performance of the Monte Carlo method to learn the time to reward with that of the Monte Carlo method to learn and perceive the time to reward in terms of the learning speed, the recovery speed from environmental changes, and the agent's performance after the learning converges in both deterministic environments and stochastic environments. Deterministic environments are considered in section 6.4.1 and stochastic environments are considered in section 6.4.2.

In order to facilitate the description of the experimental scenarios, we introduce more notation as shown in table 6.2.

6.4.1 Deterministic environments

In this subsection, we carry out experiments in environments where the length of every path is deterministic but may change over time. We first compare the training process of the two algorithms.

1. Case 1: $x_1 = x_2 = x_3 = x_4 = y_1 = y_2 = y_3 = y_4 = 10$
   The length of all paths is equal. This, however, does not mean that each

route is equal due to the topology of the graph. The shortest route in this case is (s_1, a_1) and its length is 10.

2. Case 2: $x_1 = x_2 = x_3 = x_4 = 100, y_1 = y_2 = y_3 = y_4 = 10$
   The shortest route is $(s_1, a_2) \rightarrow (s_2, a_2) \rightarrow (s_3, a_2) \rightarrow (s_4, a_2)$ and its length is 40. The difference in the time to reward between actions in one state is relatively great.

3. Case 3: $x_1 = x_2 = x_3 = x_4 = 10, y_1 = y_2 = y_3 = y_4 = 100$
   In this case, (s_1, a_1) is the shortest route and its length is 10. The difference in the time to reward between actions in one state is relatively great.

4. Case 4: $x_1 = x_2 = x_3 = x_4 = 100, y_1 = y_2 = y_3 = y_4 = 2$
   The shortest route is $(s_1, a_2) \rightarrow (s_2, a_2) \rightarrow (s_3, a_2) \rightarrow (s_4, a_2)$ and its length is 8. This case is similar to Case 2, but the difference in the time to reward between actions in one state is even greater.

5. Case 5: $x_1 = x_2 = x_3 = x_4 = 2, y_1 = y_2 = y_3 = y_4 = 100$
   (s_1, a_1) is the shortest route and its length is 2. This case is similar to Case 3, but the difference in the time to reward between actions in one state is even greater.

6. Case 6: $x_1 = 40, x_2 = 30, x_3 = 20, x_4 = 10, y_1 = y_2 = y_3 = y_4 = 10$
   In this scenario, there is no difference in the time to reward between actions in all states.

In Case 1, as figure 6.2 shows, MCTP learning learns faster than MCT learning in terms of both the time steps taken to get one reward and the learning of T values due to an increased learning rate when the estimated value is not correct, which is usually the case at the beginning of learning. Because the difference in the time to reward between actions in all states is relatively small, giving up only occurs when the learning agent chooses the second action in all first three states. The chance that this situation happens is very small because the second actions in the first three states are all suboptimal actions. Therefore giving up seldom occurs in this scenario. This explains why the two algorithms behave similarly after the learning converges. When the difference in length between routes increases, however, giving up takes effect. As figure 6.3 for Case 2 and figure 6.4 for Case 3 show, it takes MCTP learning fewer time steps to reach its

destination than MCT learning after the learning converges in both scenarios. It is worth noting that, in figure 6.3, it takes MCTP learning more time to converge in this scenario. This is because at the beginning of the learning, the learning agent is more likely to go through suboptimal paths in this scenario, which causes incorrect estimation of $T(1,2)$, $T(2,2)$ and even $T(3,2)$. Because MCTP learning has a faster learning rate when the estimated value is not correct, its estimation of $T(1,2)$, $T(2,2)$, $T(3,2)$ is further away from their true value and therefore it takes longer to correct the mistake. Comparing figure 6.3 (b.) and (d.), we can see that for MCTP learning, $T(1,2)$, $T(2,2)$ and $T(3,2)$ increase more quickly at the beginning and therefore need more time to drop back afterwards than those for MCT learning.

To further compare the performance of the two algorithms in terms of the time steps taken to reach the destination after the learning converges, we calculate the average time steps to get one reward by the two algorithms in the last 100 episodes in the above six scenarios. The results are shown in figure 6.5. In Case 6, giving up does not occur and therefore there is no difference between the two algorithms. In Case 1, giving up seldom occurs as discussed above and therefore MCTP learning is only marginally better than MCT learning. In Case 2 and Case 3, the difference in length between paths increases, giving up occurs more often and therefore MCTP learning is apparently better than MCT learning. In Case 4 and Case 5, the difference in the length between paths becomes even greater, giving up can save more time and therefore MCTP learning is much better than MCT learning.

We also do an interesting experiment using a scenario similar to the cliff-walking task [2].

1. Case 7: $x_1 = 16, x_2 = x_3 = x_4 = 100, y_1 = y_2 = y_3 = y_4 = 2$
   Like the cliff-walking task, the shortest route in this scenario is not necessarily the optimal route for a non-greedy learning agent because a non-greedy learning agent may easily enter a much worse situation if it follows the shortest route. But unlike the cliff-walking task, the learning agent in this scenario can still reach its destination and also has opportunities to go back even if it enters such a situation.

We experiment on standard Q learning, MCT learning and MCTP learning in this scenario. The settings for MCT learning and MCTP learning are the same as for previous experiments. For Q learning, the state representation is the same

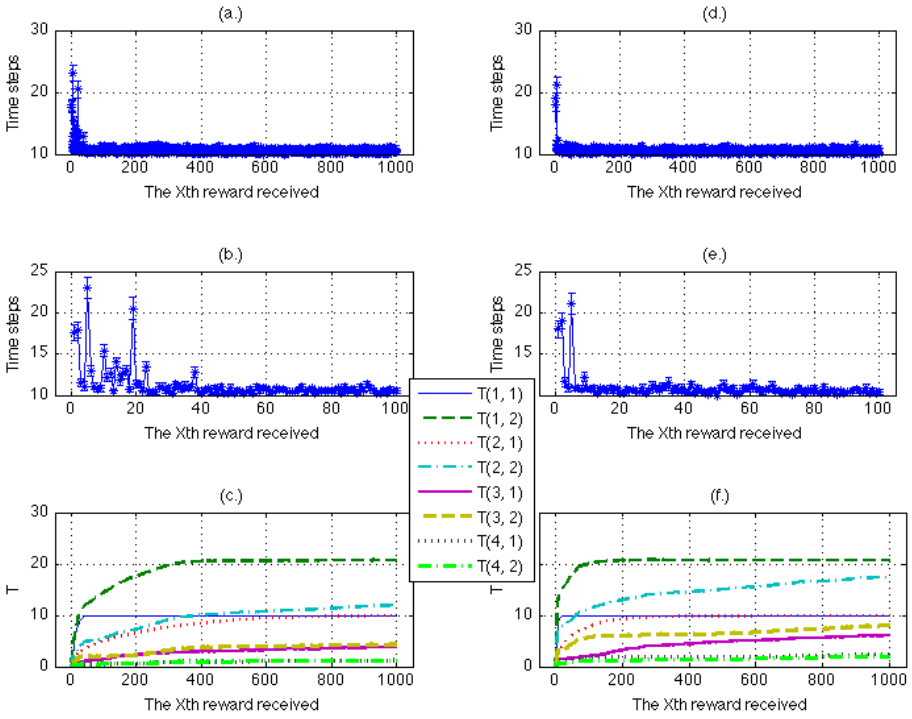

Figure 6.2: Comparison of the two algorithms during training; ; Case 1. (a.) time steps taken to get the Xth reward by MCT learning; (b.) same with (a.), but only shows the first 100 episodes; (c.) the learning of T values by MCT learning; (d.) time steps taken to get the Xth reward by MCTP learning; (e.) same with (d.) but only shows the first 100 episodes (f.) the learning of T values by MCTP learning.

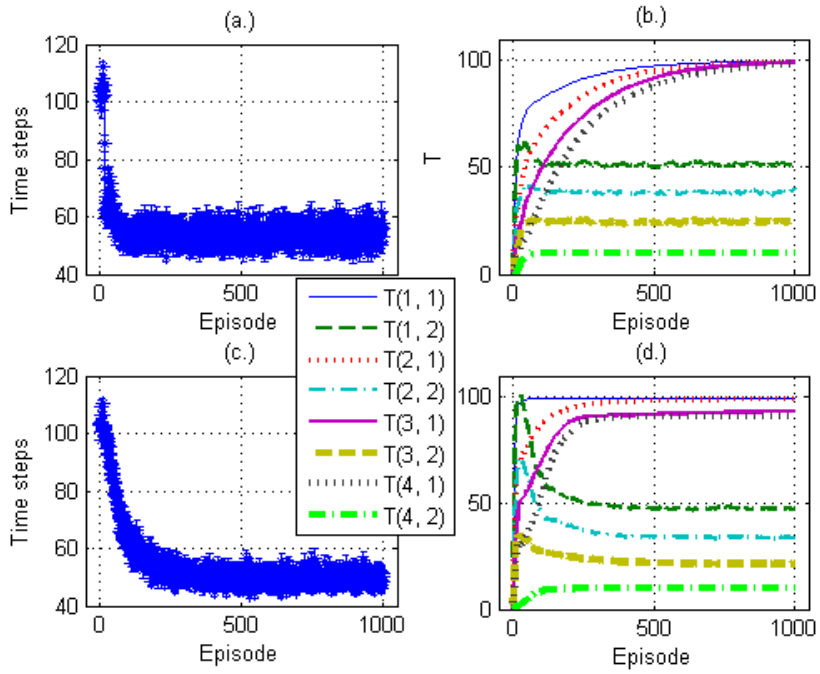

Figure 6.3: Comparison of the two algorithms during training; Case 2. (a.) time steps taken to get the Xth reward by MCT learning; (b.) the learning of T values by MCT learning; (c.) time steps taken to get the Xth reward by MCTP learning; (d.) the learning of T values by MCTP learning.

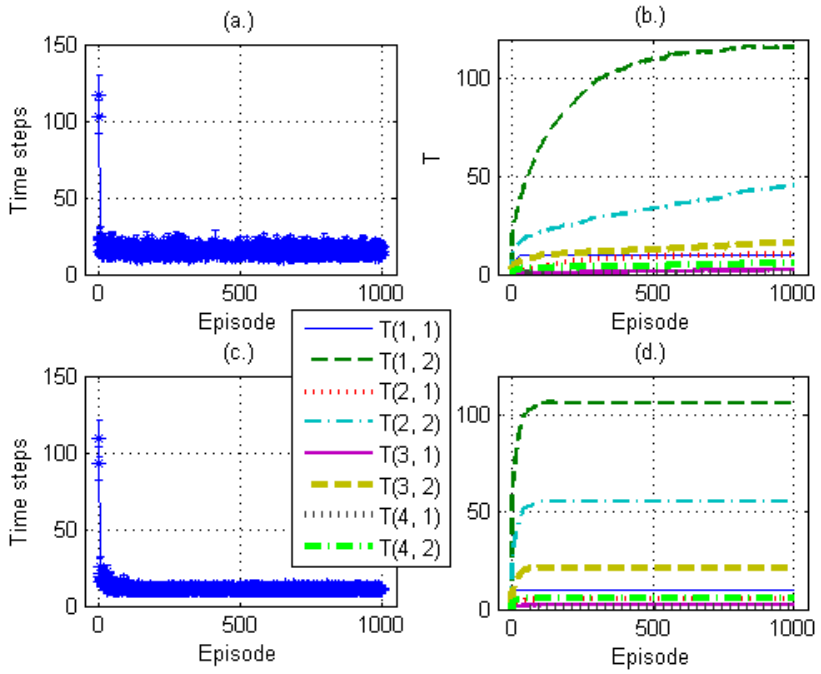

Figure 6.4: Comparison of the two algorithms during training; Case 3. (a.) time steps taken to get the Xth reward by MCT learning; (b.) the learning of T values by MCT learning; (c.) time steps taken to get the Xth reward by MCTP learning; (d.) the learning of T values by MCTP learning

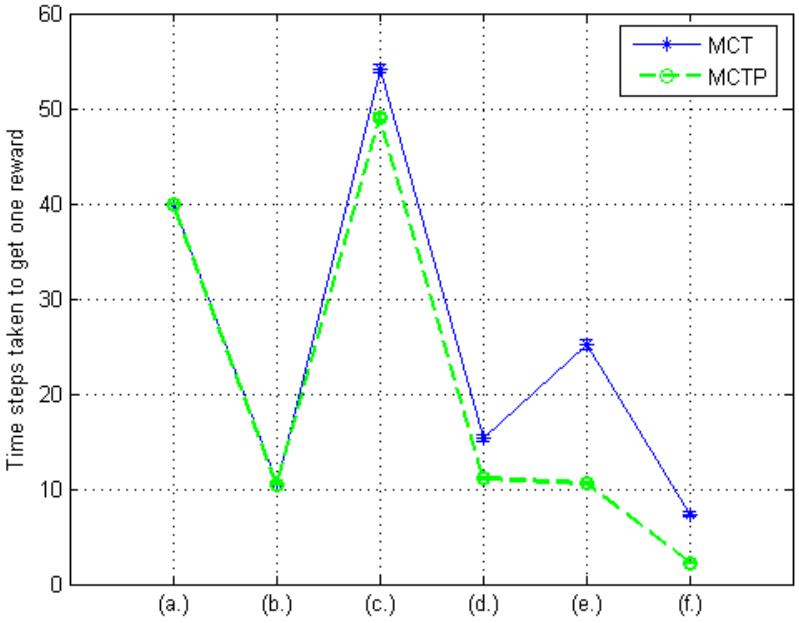

Figure 6.5: Comparison of the two algorithms in terms of the average time steps to get one reward in the last 100 episodes. (a.) Case 6; (b.) Case 1; (c.) Case 2; (d.) Case 3; (e.) Case 4; (f.) Case 5.

and the parameters are as usual ($\alpha = 0.1, \epsilon = 0.1, \gamma = 0.9$). As figure 6.6 shows, Q learning performs the worst of the three. This is because Q learning, as an off-policy learner, chooses to follow the shortest route most of time and, due to its non-greedy decision making, it is very likely that it may deviate from the shortest route and enter one path with length 100 before it reaches its destination. MCT learning performs better than Q learning. As an on-policy learner, MCT learning chooses the first action in the first state most of time and avoids the risky shortest route. MCTP learning performs the best of the three. Like Q learning, it also chooses to follow the shortest route most of time. But unlike Q learning, it would give up to the previous junction if it has found that the current path is still worse than the optimal one after it enters a path with length 100 and would therefore avoid being trapped in suboptimal paths for too long.

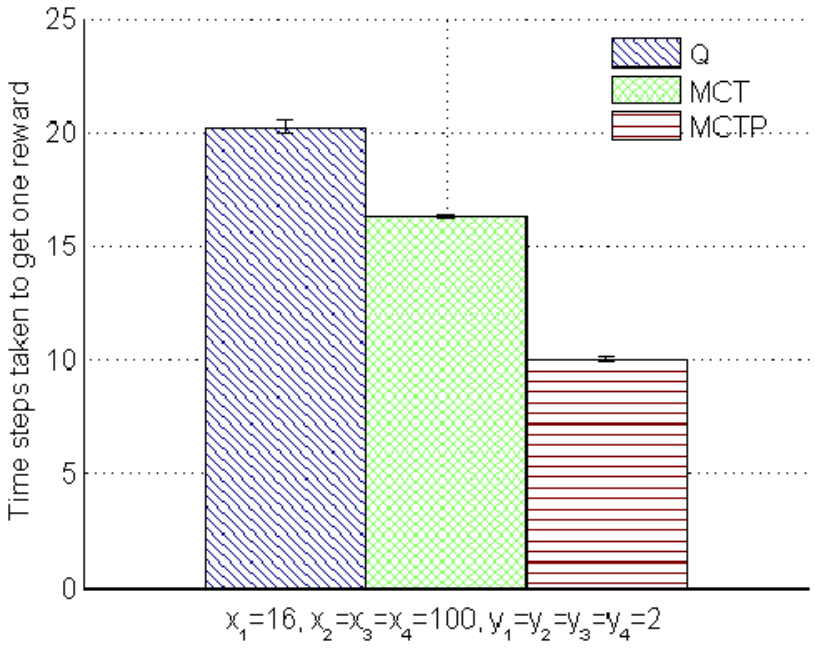

Figure 6.6: Comparison of the three algorithms in terms of the average time steps to get one reward after the learning converges; $x_1 = 16, x_2 = x_3 = x_4 = 100, y_1 = y_2 = y_3 = y_4 = 2$.

Next, we compare their performance in terms of the recovery speed from an environmental change. In this problem, there is more than one state. As mentioned previously, it would take a very long time to learn the values of all state-action pairs correctly. For limited episodes, the degree to which the learning agent has

learned may be different for different learning algorithms, which would in turn affect their performance in terms of the recovery speed from an environmental change. It is not hard to image that the better the learning agent learns the old environment the longer it may take the learning agent to unlearn the old environment and then learn the new environment. Therefore, in addition to using the same algorithm for both training and recovery, we also use a common algorithm, e.g. MCT learning, to learn the old environment first and then use the two algorithms to recover from the environmental change so that we can eliminate the influence of the training on the performance of recovery from an environmental change. We consider the following three scenarios.

1. Case 8: $x_1 = x_2 = x_3 = x_4 = 100, y_1 = y_2 = y_3 = y_4 = 10$, $x'_1 = x'_2 = x'_3 = x'_4 = 10, y'_1 = y'_2 = y'_3 = y'_4 = 100$
   At the beginning, the shortest route is $(s_1, a_2) \rightarrow (s_2, a_2) \rightarrow (s_3, a_2) \rightarrow (s_4, a_2)$. In this environment, the optimal action in the last state is the second action. The optimal action in the first three states, however, depends on the policy of the learning agent. If the learning agent chooses the second action most of time in all states, the second action in the first three states is also the optimal action. Otherwise, the first action in the first three states is the optimal action because $x_1 < y_1 + x_2; x_2 < y_2 + x_3; x_3 < y_3 + x_4$. When the environment changes, (s_1, a_1) becomes the new shortest route. In the new environment, the first action in all states is the optimal action.

2. Case 9: $x_1 = x_2 = x_3 = x_4 = 10, y_1 = y_2 = y_3 = y_4 = 100$, $x'_1 = x'_2 = x'_3 = x'_4 = 100, y'_1 = y'_2 = y'_3 = y'_4 = 10$
   This case is just the opposite to the first case. If an algorithm can cope well with the environmental change in the first case, it does not mean that it is good at handling environmental changes because it may be its special features that enable it to recover from that particular kind of environmental change very quickly. If so, however, it should perform particularly badly if a contrary environmental change occurs. This scenario tests if this is the case.

3. Case 10: $x_1, x_2, x_3, x_4, y_1, y_2, y_3, y_4, x'_1, x'_2, x'_3, x'_4, y'_1, y'_2, y'_3, y'_4$
   $\sim IntegerDiscreteUniform(1, 101)$
   The length of each path is drawn from an integer number discrete uniform distribution ranging from 1 to 101 once at the beginning and once when

the environment changes. In this scenario, the length of each path can be different and can become shorter, longer or stay the same (even though the chance of which is very small) when the environment changes. The shortest route or optimal route (these two paths can be different depending on the policy of the agent) may also stay the same or change when the environment changes. This scenario is used to test the average performance of the two algorithms.

As figure 6.7 shows, when the same algorithm is used for both training and recovery, MCTP learning recovers from the environmental change slightly quicker than MCT learning in Case 8, apparently quicker than MCT learning in both Case 9 and Case 10. When MCT learning is used for training in both experiments, however, MCTP learning recovers from the environmental change much quicker than MCT learning in all the three cases.

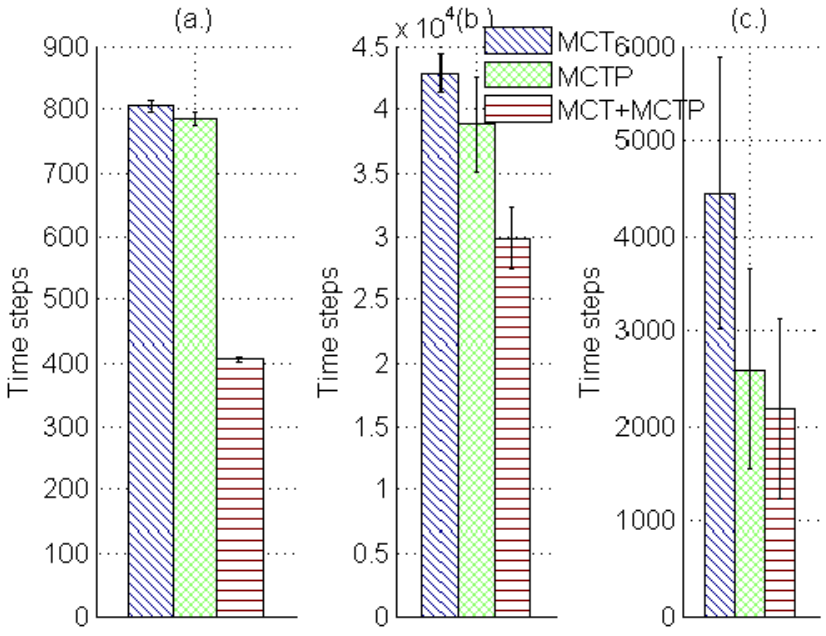

Figure 6.7: Time steps taken to recover from environmental changes. (a.) Case 8; (b.) Case 9; (c.) Case 10.

6.4.2 Stochastic environments

In this subsection, we do sets of experiments similar to those in the last subsection but in stochastic environments where the length of every path is stochastic instead of deterministic. We first compare the training process of the two algorithms.

1. Case 1: $x_1, x_2, x_3, x_4, y_1, y_2, y_3, y_4 \sim Poisson(10) + 1$
   The mean of the length of all paths is equal. This, however, does not mean that each route is equal due to the topology of the graph or that the length of all paths in one episode is the same due to the stochastic environment. The shortest route in this case is (s_1, a_1) on average and its length is 11 on average.

2. Case 2: $x_1, x_2, x_3, x_4 \sim Poisson(100) + 1; y_1, y_2, y_3, y_4 \sim Poisson(10) + 1$
   The shortest route on average is $(s_1, a_2) \rightarrow (s_2, a_2) \rightarrow (s_3, a_2) \rightarrow (s_4, a_2)$ and its length on average is 44.

3. Case 3: $x_1, x_2, x_3, x_4 \sim Poisson(10) + 1; y_1, y_2, y_3, y_4 \sim Poisson(100) + 1$
   In this case, (s_1, a_1) is the shortest route on average and its length is 11 on average. The difference in the mean of the time to reward between actions in one state is relatively great.

4. Case 4: $x_1, x_2, x_3, x_4 \sim Poisson(100) + 1; y_1, y_2, y_3, y_4 \sim Poisson(2) + 1$
   The shortest route is $(s_1, a_2) \rightarrow (s_2, a_2) \rightarrow (s_3, a_2) \rightarrow (s_4, a_2)$ on average and its length is 12 on average. This case is similar to Case 2, but the difference in the mean of the time to reward between actions in one state is even greater.

5. Case 5: $x_1, x_2, x_3, x_4 \sim Poisson(2) + 1; y_1, y_2, y_3, y_4 \sim Poisson(100) + 1$
   (s_1, a_1) is the shortest route on average and its length is 3 on average. This case is similar to Case 3, but the difference in the mean of the time to reward between actions in one state is even greater on average.

6. Case 6: $x_1 \sim Poisson(40) + 1, x_2 \sim Poisson(30) + 1, x_3 \sim Poisson(20) + 1, x_4 \sim Poisson(10) + 1; y_1, y_2, y_3, y_4 \sim Poisson(10) + 1$
   In this scenario, there is almost no difference in the mean of the time to reward between actions in all states on average. However, the actual time to reward for actions in one state can be quite different in one episode.

In Case 1, as figure 6.8 shows, similar to the results for the deterministic environment, MCTP learning learns faster than MCT learning in terms of both the time steps taken to get the xth reward and the learning of T values due to a greater learning rate when the estimated value is not correct, which is usually the case at the beginning of learning. For the same reason discussed in the last subsection, giving up seldom occurs in this scenario and therefore the two algorithms behave similarly after learning converges. When the difference in length between paths increases, however, giving up takes effect. As figure 6.9 for Case 2 and 6.10 for Case 3 show, similar to the results for the deterministic environment, it takes MCTP learning fewer time steps to reach its destination than MCT learning after the learning converges in both scenarios. It is worth noting that, in figure 6.9, similar to the results for the deterministic environment, it takes MCTP learning more time to converge in this scenario for the same reason discussed in the last subsection.

To further compare the performance of the two algorithms in terms of the time steps taken to reach the destination after the learning converges, we calculate the average time steps to get one reward by the two algorithms in the last 100 episodes in the above six scenarios. The results are shown in figure 6.11. In Case 6, giving up does not occur because the estimated time to reward for the two actions in each state is almost the same, though the actual time to reward for the two actions in each state can be quite different due to the stochastic environment unlike the deterministic scenario in the last subsection. Therefore there is almost no difference between the two algorithms. In Case 1, giving up seldom occurs as discussed above and therefore MCTP learning is only marginally better than MCT learning. In Case 2 and Case 3, the difference in the mean of the length between paths increases, giving up occurs more often and therefore MCTP learning is apparently better than MCT learning. In the last two scenarios, the difference in the mean of the length between paths becomes even greater, giving up can save more time and therefore MCTP learning performs much better than MCT learning.

We also do an experiment using a scenario similar to the cliff-walking task as what we have done for deterministic environments.

1. Case 7: $x_1 \sim Poisson(16) + 1; x_2, x_3, x_4 \sim Poisson(100) + 1;$
   $y_1, y_2, y_3, y_4 \sim Poisson(2) + 1$

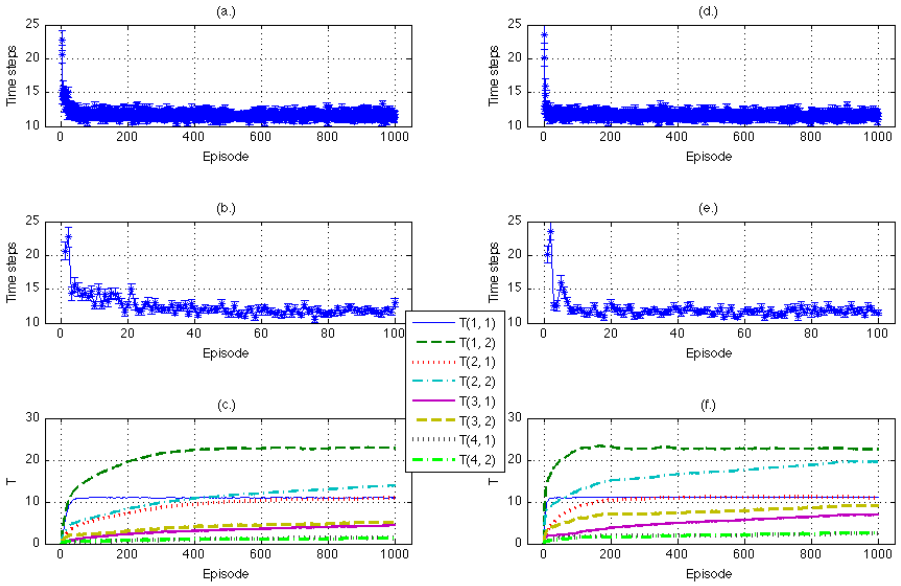

Figure 6.8: Comparison of the two algorithms during training; Case 1. (a.) time steps taken to get the Xth reward by MCT learning; (b.) same with (a.), but only shows the first 100 episodes; (c.) the learning of T values by MCT learning; (d.) time steps taken to get the Xth reward by MCTP learning; (e.) same with (d.), but only shows the first 100 episodes; (f.) the learning of T values by MCTP learning.

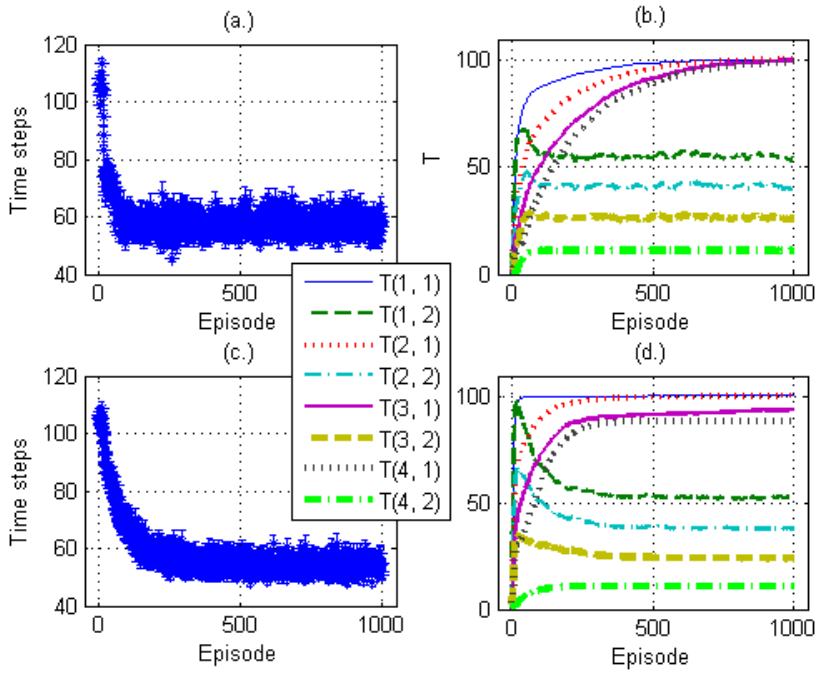

Figure 6.9: Comparison of the two algorithms during training; Case 2. (a.) time steps taken to get the Xth reward by MCT learning; (b.) the learning of T values by MCT learning; (c.) time steps taken to get the Xth reward by MCTP learning; (d.) the learning of T values by MCTP learning.

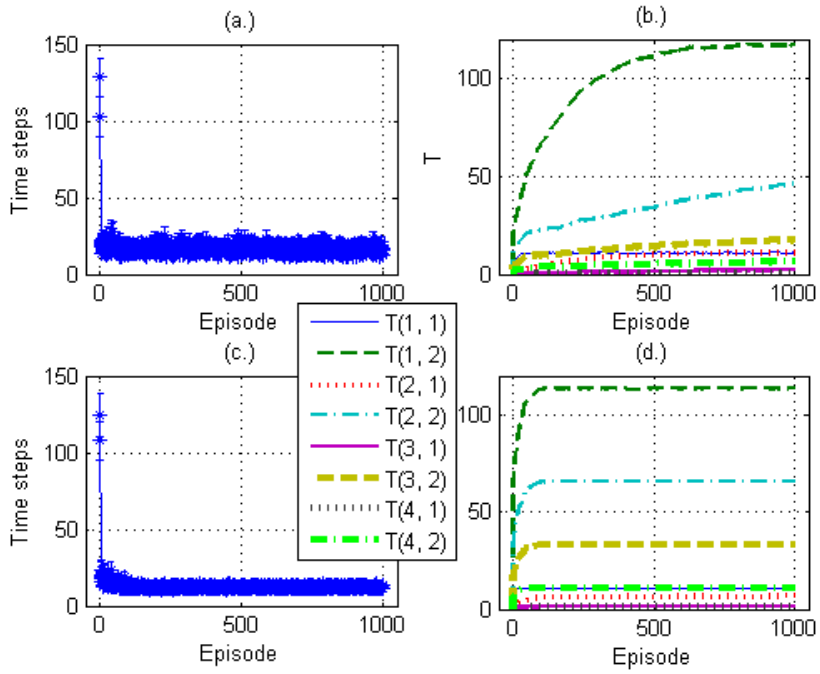

Figure 6.10: Comparison of the two algorithms during training; Case 3. (a.) time steps taken to get the Xth reward by MCT learning; (b.) the learning of T values by MCT learning; (c.) time steps taken to get the Xth reward by MCTP learning; (d.) the learning of T values by MCTP learning.

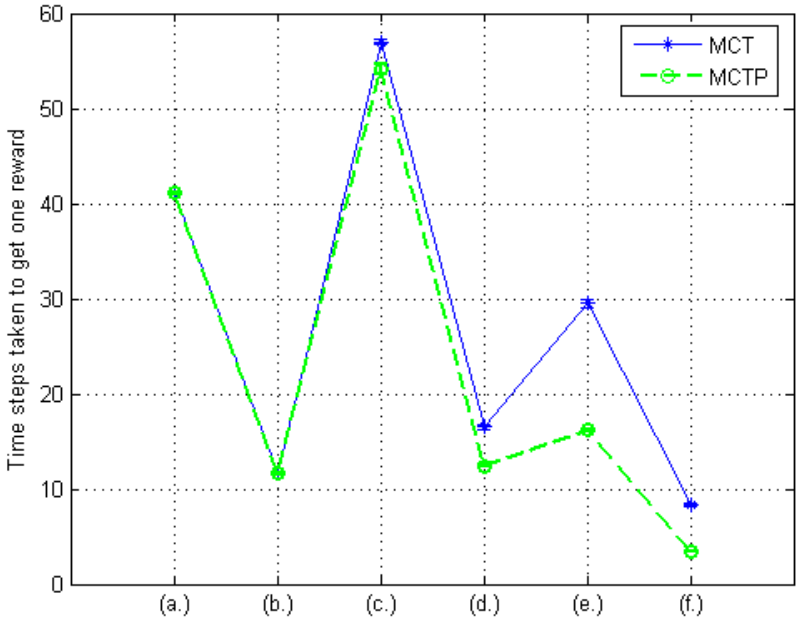

Figure 6.11: Comparison of the two algorithms in terms of the average time steps to get one reward in the last 100 episodes. (a.) Case 6; (b.) Case 1; (c.) Case 2; (d.) Case 3; (e.) Case 4; (f.) Case 5.

Similar to the last subsection, we experiment on Q learning, MCT learning and MCTP learning using this scenario with the same settings. As figure 6.12 shows, the results are similar to those in deterministic environments. Q learning performs the worst of the three, MCT learning second and MCTP learning the best for the same reason discussed in the last subsection.

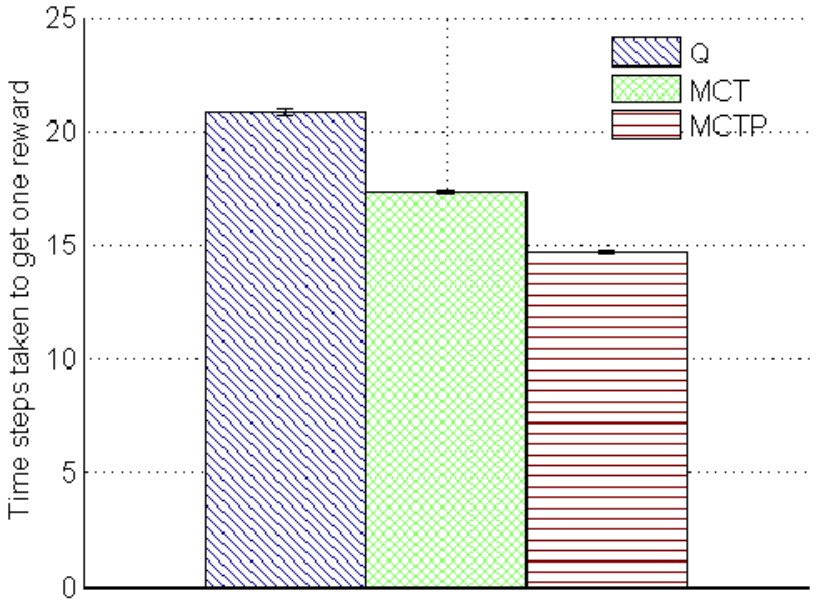

Figure 6.12: Comparison of the three algorithms in terms of the average time steps to get one reward after the learning converges; $x_1 \sim Poisson(16)+1; x_2, x_3, x_4 \sim Poisson(100)+1; y_1, y_2, y_3, y_4 \sim Poisson(2)+1$

Next, we compare their performance in terms of the recovery speed from an environmental change. For the same reason discussed in the last subsection, in addition to using the same algorithm for both training and recovery, we also use a common algorithm, e.g. MCT learning, to learn the old environment first and then use the two algorithms to recover from an environmental change so that we can eliminate the influence of the training on their performance in terms of the recovery speed from an environmental change. We consider the following three scenarios.

1. Case 8: $x_1, x_2, x_3, x_4 \sim Poisson(100)+1, y_1, y_2, y_3, y_4 \sim Poisson(10)+1$, $x'_1, x'_2, x'_3, x'_4 \sim Poisson(10)+1, y'_1, y'_2, y'_3, y'_4 \sim Poisson(100)+1$
   At the beginning, the shortest route on average is $(s_1, a_2) \rightarrow (s_2, a_2) \rightarrow$

$(s_3, a_2) \rightarrow (s_4, a_2)$. In this environment, the optimal action on average in the last state is the second action. The optimal action on average in the first three states, however, depends on the policy of the learning agent. If the learning agent chooses the second action most of time in all states, the second action in the first three states is also the optimal action on average. Otherwise, the first action in the first three states is the optimal action on average because $x_1 < y_1 + x_2$, $x_2 < y_2 + x_3$, and $x_3 < y_3 + x_4$ on average. When the environment changes, (s_1, a_1) becomes the new shortest route on average. In the new environment, the first action in all states is the optimal action on average. It is worth noting that this analysis is based only on the average case. In every episode, the actual length of each path may be quite different from its expected value. For example, in one episode, x_1 may be greater than y_1. In another episode, however, x_1 may be even smaller than y_1.

2. Case 9: $x_1, x_2, x_3, x_4 \sim Poisson(10) + 1, y_1, y_2, y_3, y_4 \sim Poisson(100) + 1$, $x'_1, x'_2, x'_3, x'_4 \sim Poisson(100) + 1, y'_1, y'_2, y'_3, y'_4 \sim Poisson(10) + 1$
   This case is just the opposite to the first case. Similar to the last subsection, this scenario is used to see if it is some special features of the algorithms that enable them to recover from that particular kind of environmental change very quickly.

3. Case 10: $x_1 \sim Poisson(\lambda_1) + 1, x_2 \sim Poisson(\lambda_2) + 1$,
   $x_3 \sim Poisson(\lambda_3) + 1, x_4 \sim Poisson(\lambda_4) + 1$,
   $y_1 \sim Poisson(\lambda_5) + 1, y_2 \sim Poisson(\lambda_6) + 1$,
   $y_3 \sim Poisson(\lambda_7) + 1, y_4 \sim Poisson(\lambda_8) + 1$,
   $x'_1 \sim Poisson(\lambda'_1) + 1, x'_2 \sim Poisson(\lambda'_2) + 1$,
   $x'_3 \sim Poisson(\lambda'_3) + 1, x'_4 \sim Poisson(\lambda'_4) + 1$,
   $y'_1 \sim Poisson(\lambda'_5) + 1, y'_2 \sim Poisson(\lambda'_6) + 1$,
   $y'_3 \sim Poisson(\lambda'_7) + 1, y'_4 \sim Poisson(\lambda'_8) + 1$
   $\lambda_1, ..., \lambda_8, \lambda'_1, ..., \lambda'_8 \sim IntegerDiscreteUniform(1, 101)$
   The length of each path is drawn from Poisson distributions with different means, which are themselves generated from an integer number discrete uniform distribution ranging from 1 to 101 once at the beginning and once when the environment changes. In this scenario, the mean of the length of each path can all be different and can become shorter, longer or stay the

same (even though the chance of which is very small) when the environment changes. The shortest route or optimal route (these two paths can be different depending on the policy of the agent) on average can also stay the same or change when the environment changes.

As figure 6.13 shows, similar to the results for the deterministic environment, when the same algorithm is used for both training and recovery, MCTP learning recovers from the environmental change slightly quicker than MCT learning in Case 8, apparently quicker than MCT learning in both Case 9 and Case 10. When MCT learning is used for training in both experiments, MCTP learning recovers from the environmental change much quicker than MCT learning in all three cases.

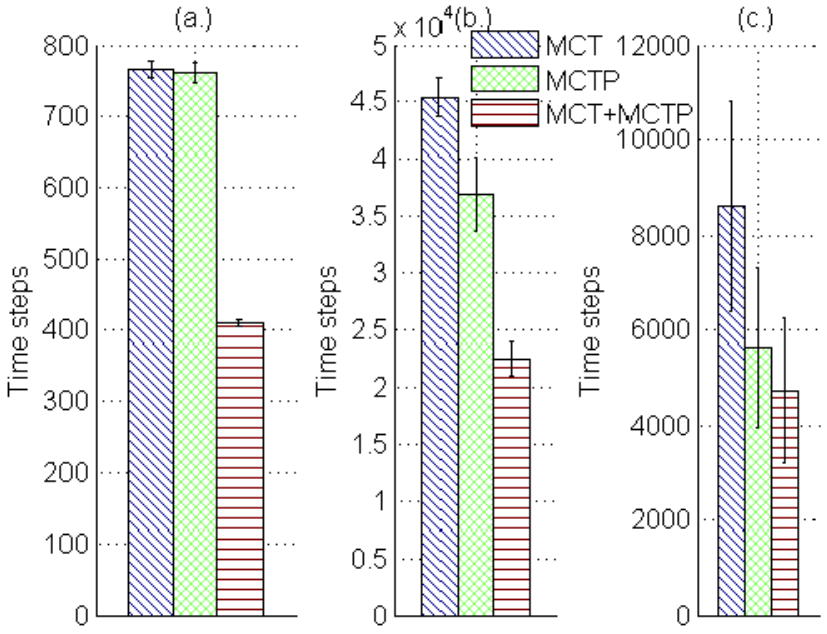

Figure 6.13: Time steps taken to recover from environmental changes. (a.) Case 8; (b.) Case 9; (c.) Case 10.

6.5 Conclusions and discussion

In this chapter, we first introduced the route finder problem, a problem with multiple states which naturally extends the time delayed n-armed bandit problem

discussed in the last chapter. We then extended the algorithms introduced in the last chapter to work with multiple states, viz. a Monte Carlo method to learn the time to reward and a Monte Carlo method to learn and perceive the time to reward. The problem is used as a test bed to compare these two algorithms. We have considered both deterministic environments and stochastic environments. In all cases, we compared the performance of these learning algorithms in terms of the learning speed and recovery speed from environmental changes, and the agent's performance after the learning converges. In all experimental scenarios, MCTP learning learns more quickly, but it does not necessarily mean that it also converges more quickly, unlike the results for the time delayed n-armed bandit problem. In regard to the recovery speed from environmental changes, when the same algorithm is used for both training and recovery, MCTP learning is better in all the cases but the difference in some cases is not great. When MCT learning is used for training in both experiments, however, MCTP learning recovers from environmental changes much quicker than MCT learning in all the test cases. After the learning converges, MCTP learning also improves the performance of the learning agent if the difference in the time to reward for actions in one state is great.

In the following subsections, we will discuss what kind of policy the algorithms learn and consider an alternative state representation where all points discretised by time steps on the route are represented as states and alternative models where the learning agent also makes decisions in non-junction states.

6.5.1 On-policy or off-policy

Like T and V learning, as introduced in the last chapter, MCT learning is also a type of on-policy learning. Like TP and VP learning, MCTP learning is also an on-policy learner when giving up has not occurred and an off-policy learner when giving up has occurred. For state-action pairs which are not given up, their time to reward is learned on-policy. For the state-action pairs which are given up, their time to reward is learned off-policy. This is because their estimated time to reward is not updated when they are given up. In other words, the policy used to make decisions contains giving up whereas the policy used to update their estimation ignores giving up. Similar to TP and VP learning, MCTP learning can also be modified to on-policy learner by updating the estimation of the given-up state-action pairs with the actual time which takes the learning agent to get a

reward since the visit to this state-action pair whether it was given up or not. Specifically, after giving up, no state-action pairs will be removed from *trackList* and the estimated time to reward for all state-action pairs in *trackList* will be updated when a reward is received. It is worth pointing out that, however, the same problems with on-policy TP or VP learning also apply to on-policy MCTP learning.

6.5.2 State representation

In the experiments, instead of considering all points discretionised by time steps on the route as states, we only represent the junctions as states since there is no decision making anywhere between two neighbouring junctions. This has dramatically reduced the state space with the same effect for both MCT and MCTP learning. For Q learning, the learning would be much slower if all time steps were represented as states. This is because the estimation of Q value is based on a bootstrap method and it takes longer to bootstrap a reward to previous states if there are more states between them when eligibility traces are not used.

6.5.3 Alternative models

When the time to reward for actions varies substantially in their values, MCTP learning improves the performance of the learning agent by giving up suboptimal actions in order to avoid longer than necessary exploration. Some may argue that we can achieve the same goal with MCT learning by augmenting the semi-MDP with time (viz. augmenting junction states with non-junction states) and allowing the learning agent to go back in non-junction states. At a junction state, the learning agent has two routes to choose. At a non-junction state, the learning agent also has two choices: to go ahead or to go back. With a greedy policy, it may work. But with a non-greedy policy, the learning agent may spend a long time going forwards and backwards in one route.

In order to test this hypothesis, we experiment on MCT learning with the choice to go back in the first deterministic scenario $x_1 = x_2 = x_3 = x_4 = y_1 = y_2 = y_3 = y_4 = 10$. The result is shown in figure 6.14 and the new algorithm does not work very well. Firstly, it takes a long time to get the first reward. This is because, at the beginning, the learning agent has equal chance to go forwards and

backwards. The learning agent has gone forwards and backwards many times before reaching its destination. Secondly, the learning is very slow. This is because there are too many state-action pairs in the problem representation. Finally, after the learning converges, it still takes much longer to reach the destination than it should. This is mainly due to non-greedy decision making. At every time step, even if the optimal action is to go forwards, the learning agent has the probability 0.05 (viz. $1 - \frac{\epsilon}{2}$) to go backwards. Within 10 time steps, the learning agent has the probability $1 - 0.95^{10} = 0.4$ to go backwards at least once. Therefore, it takes the learning agent much longer to reach the destination than it should, even after the learning converges. Furthermore, it is not hard to predict that the result would become even worse if the length of the paths were even longer, because the probability of giving up increases when the length of the paths increases.

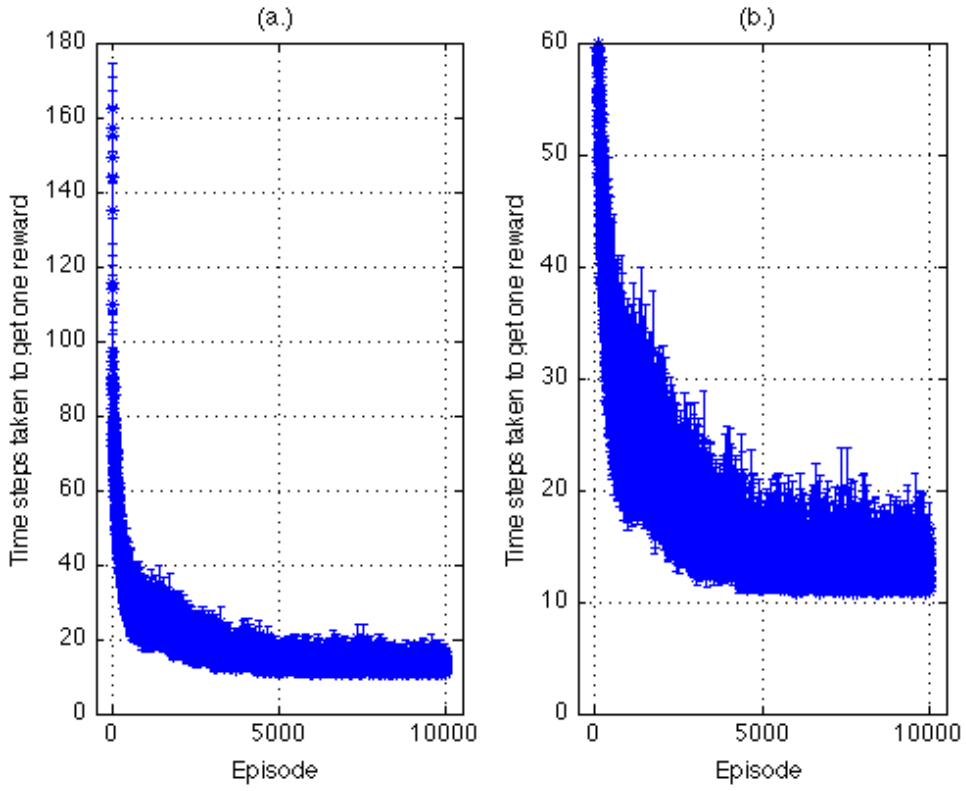

Figure 6.14: Time steps needed to get the Xth reward in a new experimental setting where junction states are augmented with non-junction states and the learning agent is allowed to go back in non-junction states. (a.) MCT learning with going both forwards and backwards allowed; (b.) the zoomed-in version of (a.).

Since augmenting the semi-MDP with time has not improved the performance of the learning agent, the following two models that we will consider do not augment the semi-MDP with time, viz. the state space only contains junction states, the same as before. Q learning with standard parameters ($\alpha = 0.1, \epsilon = 0.1, \gamma = 0.9$) is used in the experiments. In the first model, the agent also has two choices like the previous model: to go forwards or backwards when it is in the middle of a route/path, viz. between two junctions. Because non-junction states are not in the state space represented by the agent, the values of these non-junction state-action pairs are not available. Thus, the agent has to choose between going forwards and going backwards equally in non-junction states. As figure 6.15 shows, the result is even worse than that of last model. It takes longer time to get the first reward than the last model and, after the learning converges, it takes even much longer time to get a reward than the last model.

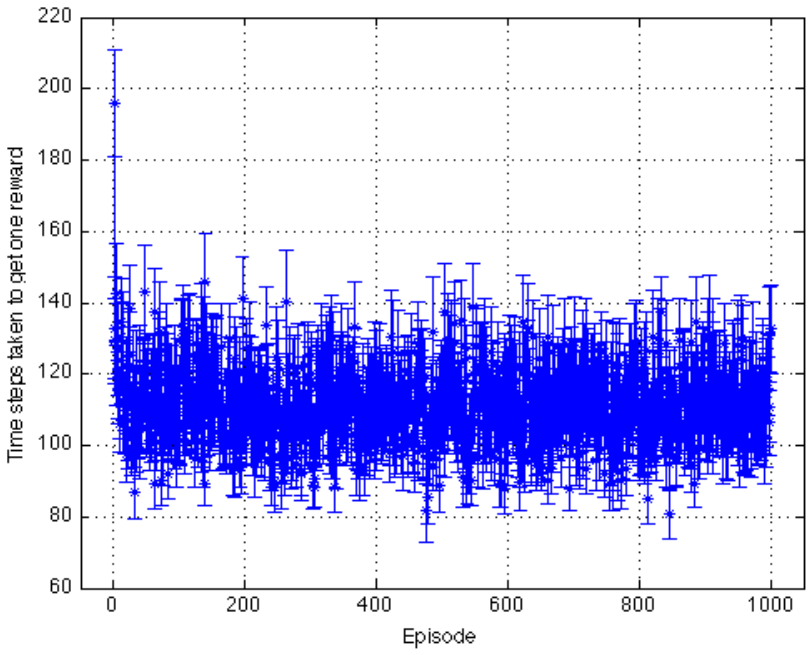

Figure 6.15: Time steps needed to get the Xth reward in a new experimental setting where the semi-MDP has not been augmented with time and the learning agent may go backwards as well as forwards in non-junction states.

In the other model, when the agent is in the middle of a route/path, it has two different choices: to go ahead one step or to give up the current route/path and then go back to the last junction and remake its decision. For the same

reason in the last model, the agent has to choose between giving up and going forwards equally in non-junction states. As figure 6.16 shows, the result is even worse than that of the last model. It takes even much longer time both to get the first reward and, after the learning converges, to get a reward than both of the last two models. From figure 6.16b, we can see that the Q value of (1,1) is unstable. When the agent receives a reward after taking the first action in the first state, $Q(1,1)$ increases. When the agent gives up after taking the first action in the first state, however, $Q(1,1)$ decreases. Therefore, the agent keeps learning and unlearning $Q(1,1)$.

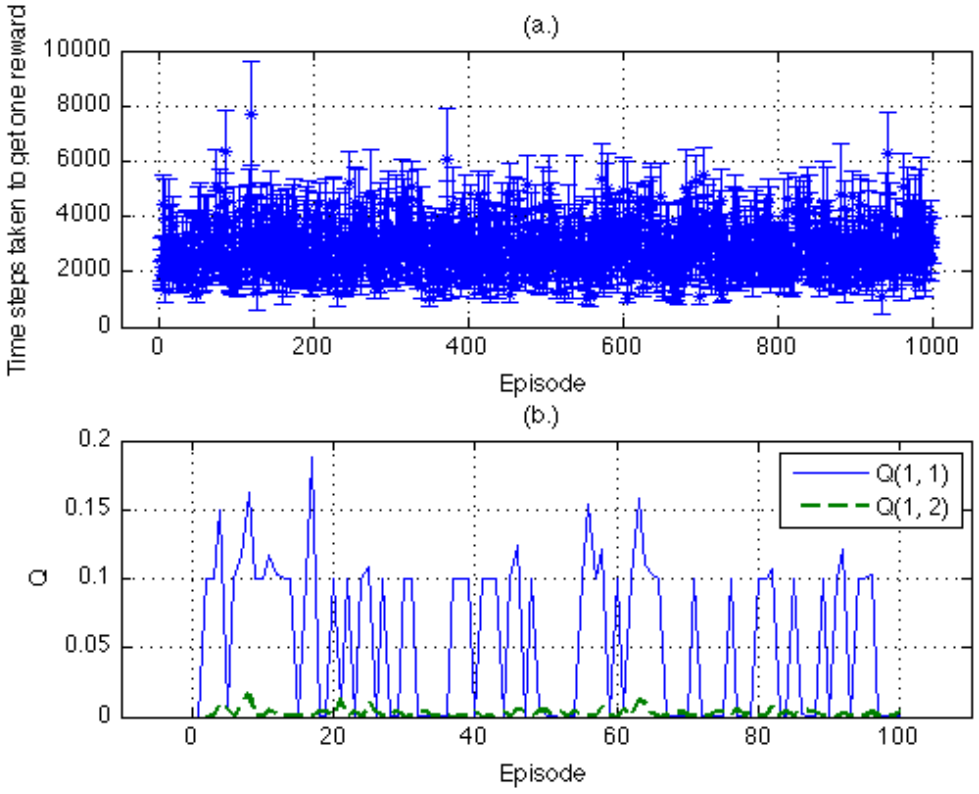

Figure 6.16: Experimental results in a new experimental setting where the semi-MDP has not been augmented with time and the learning agent may give up to the last junction state as well as go forwards in non-junction states. (a.) time steps needed to get the Xth reward; (b.) the learning of T values by Q learning.

6.5.4 Why Monte Carlo methods

As mentioned in section 3.3, the estimated mean is non-stationary during the learning period for bootstrap methods in an environment with multiple states unless a full dynamic programming backup is used. The non-stationary nature of the estimated mean will lead to a biased estimation of the variance in the value function [68], which in turn affects the performance of methods relying on unbiased statistics [69]. For this reason, we have proposed a Monte Carlo method with time perception, a non-bootstrap method, to solve the route finder problem instead of a bootstrap method, e.g. Q learning.

Chapter 7

Summary and conclusions

This concluding chapter summarises the motivation, fundamental ideas, justification for key decisions, and the main results and contributions of the research in section 7.1 (using non-technical language and omitting equations). The limitations of the research are pointed out in section 7.2 and ideas for future work are discussed in section 7.3.

7.1 Summary of the research

This PhD research is part of an ongoing research programme. The main motivation for this ongoing research shall now be outlined.

Classical value estimation reinforcement learning algorithms do not perform very well in dynamic environments. On the other hand, the reinforcement learning of animals is quite flexible: they can adapt to dynamic environments very quickly and deal with noisy inputs very effectively. Anyone who has ever learned to ride a bicycle or drive a car knows how fast humans can learn a new complex task. Similarly, dogs can perform new and very complex tasks, such as rescuing buried people during earthquakes and guiding blind people, after some training. One feature that may contribute to animals' good performance in dynamic environments is that they learn and perceive the time to reward which can be used to detect changes in the environment and decide when it should give up the current action in order to avoid longer than necessary exploration. When a change is detected, animals can respond specifically to and recover from it quickly. Motivated by this feature of animal learning, the ongoing research aims to build a biologically plausible neuron model that is capable of implementing reinforcement

learning and of perceiving the time and above all capable of adapting to dynamic environments quickly.

As part of the ongoing research, this PhD research has first explored the possibilities of using biologically plausible neuron models to implement reinforcement learning. Currently, reinforcement learning is mainly implemented by abstract models. For animals, however, neurons are the only computing units. Therefore, the reinforcement learning of animals must be implemented by neurons. Furthermore, neurons communicate by means of a sequence of short electrical pulses, the so-called spikes or action potentials in the biological neural system. Spiking neuron models emphasise that the timing of these spikes carries information and can be used for computation, which is potentially more powerful and can contain more information than traditional firing rate models. For reinforcement learning, timing is also very important. Rewards received immediately are considered more important than those received after a long time. Rewards received after a state-action pair are usually credited to the state-action pair, whereas rewards received before a state-action pair are usually not credited to the state-action pair. In addition, the dynamics of spiking neurons may be well suited to model reinforcement learning in dynamic environments. We have successfully used spiking neurons to implement various phenomena of classical conditioning, the simplest form of animal reinforcement learning in dynamic environments, and also pointed out a possible implementation of instrumental conditioning and general reinforcement learning using similar models.

Instead of implementing the approach of time perception with low-level biological models first and then testing whether or not it works, we decided to first study the details of time perception and test its effectiveness by using abstract reinforcement learning models and a perfect clock. Implementing it with low-level biological models would have caused the result to be implementation dependent. If it does not work, it is hard to determine whether the problem is because of the biological implementation or because of the principle (the approach itself). Besides, for the approach itself, there are still many questions to answer, many problems to address, and many details to investigate. These questions are much easier to address with abstract models than with low-level biological models.

Regarding detecting changes in the environment, what should be learned in order to detect changes in the environment? Is learning only the mean of the time to reward sufficient? When the amount of reward may also change, it is obvious

that the learned time information cannot detect changes in the amount. In this case, what should the learning agent learn to detect both changes in the time to reward and changes in the amount of reward? On the other hand, even if the amount of reward does not change, learning the mean of the time to reward alone cannot detect changes in the time to reward immediately (in one trial) when the environment is stochastic. In this case, are there any ways to detect changes in the time to reward immediately? Furthermore, after a change in the environment is detected, what should the learning agent do in order to recover from the change quickly? Regarding giving up a suboptimal action to avoid longer than necessary exploration, how can the agent decide whether or not to give up the current action? If it decides to give up the action, what state should it give up to? After giving up, what action should the agent choose?

With regard to detecting changes in the environment, we found that learning the mean of the time to reward alone is not enough. When the amount of reward does not change, the estimated mean of the time to reward cannot detect changes in the time to reward immediately (in one trial) when the environment is stochastic. It needs many trials to detect changes in a stochastic environment because it has to compare the mean of the values in recent several trials with the mean of the values in several trials before recent several trials. On the other hand, although we can use the estimated distribution of the time to reward to detect changes in the time to reward immediately (in one trial) even when the environment is stochastic, learning the distribution of a random variable is a non-trivial task in its own right. Based on Chebyshev's inequality and Cantelli's inequality, we decided to learn the variance of the time to reward as well and then to use it together with the estimated mean of the time to reward to detect changes in the time to reward. When the amount of reward may also change, another problem arises: learning the time to reward cannot detect changes in the amount of reward. Admittedly, the agent can learn both the mean and the variance of the time to reward to detect changes in the mean of the time to reward, and can learn both the mean and the variance of the amount of reward to detect changes in the amount of reward. This method, however, separates changes in the time to reward from changes in the amount of reward. From the viewpoint of the learning agent, however, the learning agent will not consider the environment as changed if its optimal target based on which it makes decisions,

e.g. the discounted reward/return, has not changed, though in reality the environment may have changed, e.g. the time to reward and the amount of reward have both changed in such a way that the discounted reward remains the same. Furthermore, using different criteria to detect environmental changes and make decisions may cause problems because their learning may not be synchronised. Suppose that the agent uses the discounted reward to make decisions and uses the time to reward to detect environmental changes. The learning of the time to reward may have converged, which leads to a decrease in the learning rate. However, the learning of the discounted reward probably has not converged. If so, it would take a long time to learn the true mean of the discounted reward because the learning rate is decreasing. Therefore, in our algorithms, only the mean and the variance of the optimal target are learned and then used to detect environmental changes and to make decisions.

When a change in the environment is detected, the learning rate is increased in order to learn the change quickly. Otherwise, the learning rate is decreased gradually towards 0 in order for the estimated mean to converge to its true mean. If the learning agent has detected that a suboptimal action has improved and can potentially become the optimal action in the new environment, the exploration rate (e.g. ϵ for ϵ-greedy) is also increased in order to increase the chance that the suboptimal action is visited. Otherwise, the exploration rate is decreased gradually towards its minimum value in order to reduce the cost of exploring suboptimal actions. The results of our experiments using two real-world problems show that, increasing the learning rate when a change in the environment is detected and increasing the exploration rate when a suboptimal action has improved and can potentially become the optimal action, have effectively improved the learning speed during training and the recovery speed when the environment changes in most of the experimental scenarios. In addition, the experiments show that decreasing the learning rate gradually towards 0, when a change in the environment has not been detected, enables the estimated mean to converge to the true mean even in stochastic environments and also enables the learning agent to find the optimal action when the mean of values (e.g. the time to reward) for actions in one state is very close in stochastic environments.

With regard to giving up a suboptimal action to avoid longer than necessary exploration, we found that the learning agent needs to compare the current action with other actions in order to find whether or not the current action is the optimal

one and when to give up the current action. Optimality is relative rather than absolute. Only when the agent finds that the current action is still worse than the optimal one should it give up this time's exploration of the action. Furthermore, the comparison only makes sense when the actions used to compare with the current action has not been overestimated. Otherwise, the current action can still be the optimal one. Here, we also use the estimated mean and variance to tell whether or not an action has been overestimated. When the amount of reward for actions is the same and does not change, the time to reward alone can be used to decide if an action is optimal, if an action has been overestimated, and when to give up a suboptimal action. When the amount of reward for actions is not the same, even though it does not change, learning time to reward alone is not enough to decide when to give up the current action. Obviously, the greater a reward it is worth waiting longer for. The learning agent may either learn the time to reward and the undiscounted reward or learn the optimal target which implicitly contains the time information (e.g. discounted reward) and the undiscounted reward. In addition, the learning agent should not be allowed to give up the current action until the undiscounted reward has been correctly learned. When the amount of reward for actions may also change, however, even if the undiscounted reward for one action has been correctly learned, the learning agent should also occasionally explore the amount of reward for one action, viz. not giving up the action, just in case the amount of its reward has changed. This is because only after the learning agent receives the reward can it know the amount of the reward. Before it knows the amount of the reward, it cannot get the time after which the current action is still worse than the optimal one, viz. decide when to give up.

When the learning agent has found that the current action in one state is still worse than the optimal one, longer exploration is unnecessary and therefore the learning agent may benefit from giving up this time's exploration of the action. This, however, is not optimal. For simplicity, suppose that the amount of reward for actions is the same and does not change so that we can just use the estimated time to reward for discussion. When the learning agent can go back without any cost and the actual time to reward for the current action in one state is less than twice the estimated time to reward for the reference action in the state, it is better not to give up this action because it can get the reward earlier by staying in the current action than by giving up the action and then choosing the reference action in the state, which would take twice the estimated time to reward for the

reference action on average. Therefore, the learning agent will not give up unless the estimated time to reward for the current action in one state is more than twice the estimated time to reward for the reference action in the state. It is apparent that the check is only useful when the estimated time to reward for the current action has been correctly learned and the estimated time to reward for the reference action in the state has not been underestimated. If these conditions are satisfied, most of the time, the learning agent only explores the current action the estimated time to reward for the reference action in the state; But at times, it explores the current action twice the estimated time to reward for the reference action in the state, just in case the time to reward for the current action has become less than twice the estimated time to reward for the reference action where it is better not to give up the current action. When there is a cost for the learning agent to give up, the threshold should be twice the estimated time to reward for the reference action plus the cost. For example, in the route finder problem, the cost of giving up and returning to a junction is just the same as that of coming here from the junction. So, the threshold for giving up should be triple the estimated time to reward for the reference action or state-action pair. When the amount of reward for actions is not the same we can still calculate the optimal time to give up. The results of our experiments using two real-world problems show that, when the difference in the time to reward or the discounted reward between actions in one state is great, this method of giving up to avoid longer than necessary exploration, has dramatically improved the performance of the learning agent after learning. On the other hand, when the difference in the time to reward or the discounted reward between actions in one state is small, giving up usually does not occur and therefore this method has little, if any, negative influence on the performance of the learning agent after learning. Furthermore, we also demonstrated that, some obvious alternative approaches that may be used to avoid longer than necessary exploration, do not work very well.

In addition, we also conducted experiments on the algorithms using a problem with a time limit or an energy budget. We have used a simple scenario to demonstrate that learning and monitoring the time to reward can improve the learning performance when the time limit or an energy budget of the agent is stochastic.

In summary, this PhD research contributes to the ongoing research by exploring the possibilities of using biologically plausible neuron models to implement

reinforcement learning, by investigating the details of possible implementations of time perception, and by exploring situations where the learned time information can be used to improve the performance of the learning agent in dynamic environments. In addition, this PhD research will also contribute to the understanding of animal behaviour and learning, and shed light on how animals use the learned time information to adapt to dynamic environments quickly.

7.2 The limitations of the research

We use second order statistics to detect changes in the environment. When the actual value/return of (s, a) is outside $Q(s, a) \pm k\sqrt{Q\_var(s, a)}$ where $Q(s, a)$ and $Q\_var(s, a)$ is the estimated mean and variance of the value of the state-action pair (s, a) and $k \geq 0$, we consider that the value of (s, a) has changed. Although this method has the potential to detect changes in the environment with only one trial even in a stochastic environment, it does have some limitations. Firstly, however large the window is except infinity, there is no guarantee that all data from the same distribution are within the window for some distributions, e.g. normal distribution. This means that a false detection of a change is unavoidable. Therefore, the policy used to handle environmental changes needs to accommodate these data, e.g. responding incrementally rather than abruptly. Secondly, when the value changes not very greatly in a stochastic environment, e.g., the new value is still within $Q(s, a) \pm k\sqrt{Q\_var(s, a)}$, this method cannot detect the change. But, if a change in the environment is very small, it may not be necessary to respond specifically to the change since classical value estimation reinforcement learning algorithms may be able to handle the change very well. Thirdly, if the mean does not change but the variance increases, this method may still consider the environment changed. In addition, the detection rule depends on the estimated variance. Unfortunately, however, the estimated variance tends to become bigger than the actual new variance when the environment changes. When the environment changes, two things contribute to the variance, viz., the change in the environment (more precisely, the change in the mean) and the variance of the new environment. The overestimated variance may cause the changed actual value to fall within the estimated mean plus and minus k standard deviations, even though the learning agent has not recovered from the environmental change, viz. the estimated mean is still incorrect. In order to reduce the influence

of a change in the environment on the estimation of variance, a smaller learning rate is used to learn the variance when the environment changes. Admittedly, it would be ideal to eliminate the influence of a change in the environment on the estimation of variance completely and only learn the variance of the new environment. Unfortunately, however, it seems impossible because the mean of the new environment is unknown. Lastly, k cannot be too small or too big. A big k can reduce the chance of a false detection of environmental changes but it may increase the chance of failing to detect environmental changes, viz. less sensitive to a change in the environment. On the other hand, a small k can detect even small environmental changes, but it is more likely to make a false detection of environmental changes, viz. too sensitive to a change in the environment.

To apply the giving up strategy (giving up to avoid longer than necessary exploration), the rule of the problem should allow giving up without losing too much. In a chess match, for example, if it is allowed to retract its moves without limitations, the learning agent can give up back to any previous move in one episode. If it is allowed to ask for a draw and then restart the match, the learning agent can give up to the beginning. On the other hand, however, if the learning agent can neither retract its moves nor ask for a draw, the giving up strategy cannot be applied to the problem.

In addition, as mentioned in section 3.3, the estimated mean is non-stationary during the learning period for bootstrap methods in an environment with multiple states unless a full dynamic programming backup is used. The non-stationary nature of the estimated mean will lead to a biased estimation of the variance in the value function [68], which in turn affects the performance of methods relying on unbiased statistics [69].

We have to point out that our method still cannot solve the shortcut problem mentioned in subsection 3.5.3. On the other hand, it will not make things worse because it only gives up the current action when it is still worse than the optimal one and will not give up actions that can potentially be the optimal one. In addition, it can also be incorporated with exploration bonus and other similar methods which are specifically designed to solve this kind of problem.

We have mainly experimented with problems where there is only one reward. In terms of problems with multiple rewards, there are two cases. The first case is that each episode still ends when any of the rewards is received. In fact, we have discussed this case in the time delayed n-armed bandit problem. When the

amount of reward for arms is not exactly the same, the problem belongs to this case and we use VP/VPWG learning to solve the problem. VP/VPWG learning, when extended to multiple state problems, can also be used to solve this case with multiple states. The other one is that each episode does not necessarily end when one reward is received. As discussed in subsection 2.2.5, this case causes the credit structuring problem. Improper reward assignment will lead to slow learning or even undesired behaviours. For example, in a chess game, if the learning agent is given a reward both when it wins and when it achieves a subgoal such as taking a piece from its opponent. It may find a way to maximise its rewards by taking the opponent's pieces even at the price of losing the game. Nevertheless, the method of detecting and responding to the environmental change should still work in this setting without any modification. For the method of giving up to avoid longer than necessary exploration, however, it will not work in this setting without a significant modification. Finally, though we have only considered problems with discrete time steps, the methods should work for problems with continuous time as well.

7.3 Future work

As mentioned previously, this PhD research is part of ongoing research which aims to build a biologically plausible neuron model that is capable of implementing reinforcement learning and of adapting to dynamic environments quickly. Since we have demonstrated the possibilities of implementing reinforcement learning using biologically plausible neuron models and the possibilities of using the learned time information to improve the performance of the learning agent in dynamic environments, the next big step is to build a biologically plausible neuron model that can do what we have done with the abstract reinforcement learning models.

Firstly, we need to build a biological clock to perceive and measure time. The clock needs to have an enormous dynamic range because a reward may occur in a short time or in a long time. The possible candidates include accumulators and a bank of filters sensitive to a bank of intervals. Secondly, we need a neuron model that can learn the time to reward. Thirdly, we need a neuron model that can also learn the variance of values in addition to the mean of values. Finally, we also need to implement something similar to neuron modulators such as neocortical acetylcholine (ACh) and neocortical norepinephrine (NE) which

can detect whether the actual value is outside its estimated mean plus and minus twice or triple standard deviations, and accordingly boost or reduce the rate of learning.

As well as a biological implementation, there are other interesting topics worth consideration. In each problem, we have only experimented with relatively small-scale problems. It would be interesting to test the scalability of the algorithms by experimenting with more complex large scale problems. One good candidate is a more general and complex route finder problem, which extends the simple route finder problem introduced in chapter 6. As figure 7.1 shows, the routes are formed using a 2-D grid. The problem itself is highly scalable and can be scaled to an nxn grid (where n is any integer no less than 2). In fact, figure 7.1 (a.) is just the 2-armed bandit problem we introduced in chapter 5. In addition, there are circles in the graph and therefore the learning agent can go back to a state in one episode without giving up. Lastly, the position of the destination/reward can also be changed in addition to the length of paths.

Large scale problems pose some difficulties. We cannot simply use a lookup table to store the information mapping from state-action pairs to their values because the table would become too large. Another problem is that many state-action pairs are seldom visited and some of them may even never have been encountered before. Thus, generalisation is needed to avoid storing a great quantity of data and to predict the values of the states or state-action pairs not experienced previously. Possible candidates for generalisation purpose include neural networks, fuzzy logic and local memory-based methods. Since our ultimate goal is to implement our algorithms using biologically plausible neuron models, we will also attempt to use biologically plausible neuron models to implement generalisation in the future. Secondly, there may be more than two actions in one state. Therefore, it is better to use a softmax method to make non-greedy decision making instead of ϵ-greedy in this case so that only the suboptimal actions which can potentially become the optimal one get more chance of being explored.

Furthermore, we have not systematically carried out comparative experiments on different parameter settings. It would be interesting to study the performance of learning algorithms in different environments with different parameters. Finally, we have only experimented on non bootstrap methods such as Monte Carlo methods because the variance can be easily learned in a non bootstrap style. It

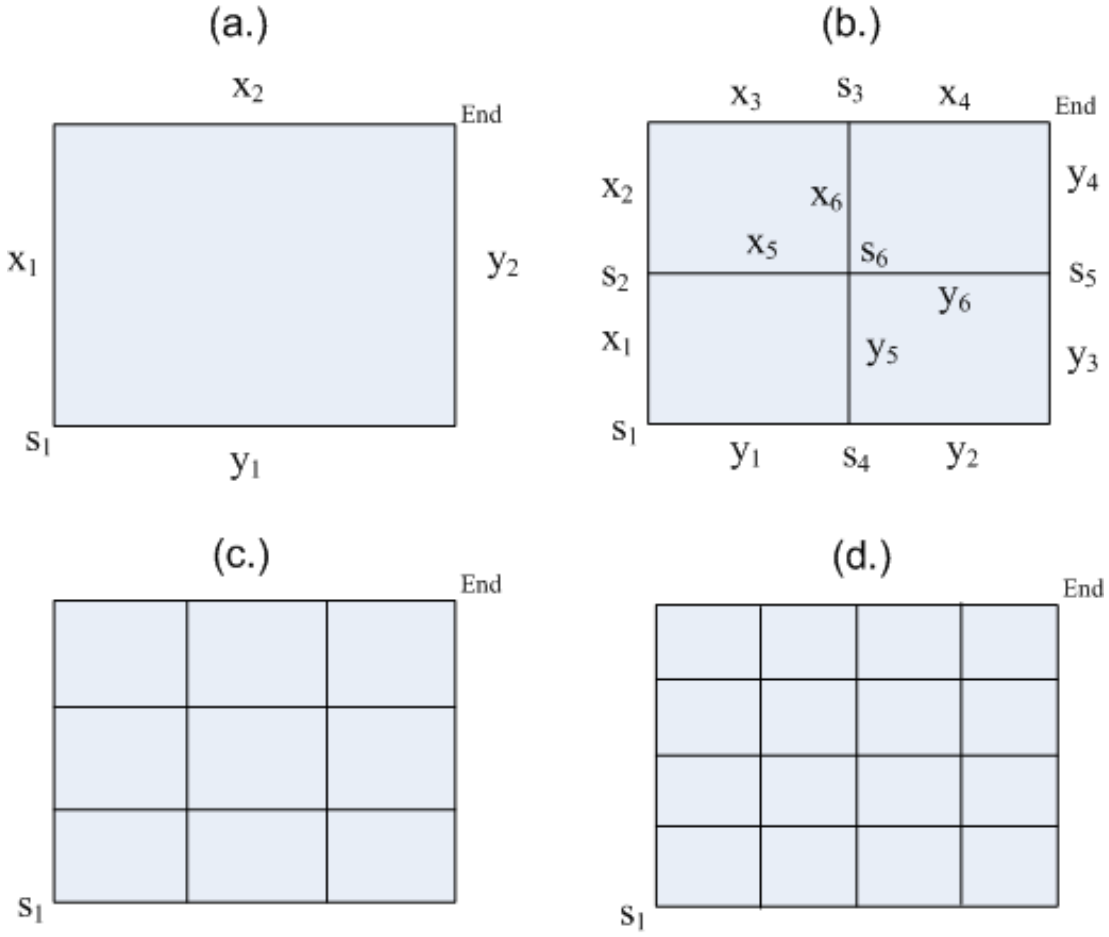

Figure 7.1: Illustration of a scalable general route finder problem. The learning agent always starts at s_1 and tries to reach the destination in the shortest time. s: states, the junctions of routes; x, y: the length of paths; the states and lengths of paths in (c.) and (d.) have not been marked due to the proliferation of states and paths available. There is only 1 state in (a.), 6 states in (b.), 13 states in (c.) and 22 states in (d.).

would be interesting to integrate learning the mean and variance with bootstrap methods, e.g. Q learning. There are two possibilities in this regard. The first one is still to use Monte Carlo methods to learn the mean and variance of values and then use the eligibility trace to update Q values with appropriate learning rates when a reward is received. The second one is to learn the mean of the value using Q learning and to learn the variance of the Q value using the following learning rule as shown in

$$\begin{aligned} Q\_var(s,a) \leftarrow & Q\_var(s,a) + \alpha_2\{[r + \gamma \max_{a'} Q(s',a') - Q_{old}(s,a)] \\ & [r + \gamma \max_{a'} Q(s',a') - Q(s,a)] - Q\_var(s,a)\}. \end{aligned} \tag{7.1}$$

where s is the current state, a is the action taken in s, s' is the next state after taking action a, r is the reward received after taking a, α_2 is the learning rate used to learn the variance, and γ is the discount factor, and $Q_{old}(s,a)$, $Q(s,a)$ and $Q\_var(s,a)$ are respectively the last and the current estimation of the mean of the value of (s,a) and the estimation of the variance of the value of (s,a). After both the mean and variance of Q values are learned, they can then be used to detect environmental changes and then to respond quickly to the change as well as to decide when to give up the current action to avoid longer than necessary exploration just like VP and MCTP learning.

As mentioned in section 3.3, however, the estimated mean is non-stationary during the learning period for bootstrap methods in an environment with multiple states unless a full dynamic programming backup is used. The non-stationary nature of the estimated mean will lead to a biased estimation of the variance in the value function [68], which in turn affects the performance of methods relying on unbiased statistics [69].

Bibliography

[1] L. P. Kaelbling, M. L. Littman, and A. W. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.

[2] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1998.

[3] D. Michie and R. A. Chambers. Boxes: An experiment in adaptive control. In E. Dale and D. Michie, editors, *Machine Intelligence 2*, pages 137–152. Oliver and Boyd, 1968.

[4] K. S. Narendra and R. M. Wheeler. Decentralized learning in finite Markov chains. *IEEE Transactions on Automatic Control*, AC31(6):519–526, 1986.

[5] S. P. Singh and R. S. Sutton. Reinforcement learning with replacing eligibility traces. *Machine Learning*, 22:123–158, 1996.

[6] R. E. Bellman. *Dynamic Programming*. Princeton University Press, 1957.

[7] C. J. C. H. Watkins. *Learning from Delayed Rewards*. PhD thesis, Cambridge University, 1989.

[8] G. A. Rummery and M. Niranjan. On-line q-learning using connectionist systems. Technical report, Cambridge University Engineering Department, 1994.

[9] Andrew W. Moore and Christopher G. Atkeson. Prioritized sweeping: Reinforcement learning with less data and less time. In *Machine Learning*, pages 103–130, 1993.

[10] A Kacelnik and M Bateson. Risky theories - The effects of variance on foraging decisions. *AMERICAN ZOOLOGIST*, 36(4):402–434, SEP 1996.

Symposium on Risk Sensitivity in Behavioral Ecology, at the Annual Meeting of the American-Society-of-Zoologists, ST LOUIS, MO, JAN 04-08, 1995.

[11] Y Niv, P Dayan, and JP O'Doherty. Decision making: Neural prediction errors show risk sensitivity. In *Computational and Systems Neuroscience*, 2008.

[12] J. S. Young. Discrete-trial choice in pigeons: Effects of reinforcer magnitude. *J. Exp. Anal. Behav*, 35:23–29, 1981.

[13] D. A. Case, P. Nichols, and E. Fantino. Pigeon's preference for variable-interval water reinforcement under widely varied water budgets. *J. Exp. Anal. Behav*, 64:299–311, 1995.

[14] K. D. Waddington, T. Allen, and B. Heinrich. Floral preferences of bumblebees (bombus edwardsii) in relation to intermittent versus continuous rewards. *Animal Behaviour*, 29:779–784, 1981.

[15] K. C. Clements. Risk-aversion in the foraging blue jay, cyanocitta cristala. *Animal Behaviour*, 40:182–195, 1990.

[16] J. E. Mazur. Theories of probabilistic reinforcement. *J. Exp. Anal. Behav.*, 51:87–99, 1989.

[17] K. D. Waddington. Bumblebees do not respond to variance in nectar concentration. *Ethology*, 101:33–38, 1995.

[18] D. W. Stephens. The logic of risk-sensitive foraging preferences. *Animal Behaviour*, 29:628–629, 1981.

[19] T. Caraco, W. U. Blanckenhorn, G. M. Gregory, J. A. Newman, G. M. Recer, and S. M. Zwicker. Risk-sensitivity: Ambient temperature affects foraging choice. *Animal Behaviour*, 39:338 – 345, 1990.

[20] Robert H. Macarthur and Eric R. Pianka. On optimal use of a patchy environment. *The American Naturalist*, 100(916):603–609, 1966.

[21] E. L. Charnov. Optimal foraging: the marginal value theorem. *Theoretical Population Biology*, 9:129–136, 1976.

[22] D. J. Howell and D. L. Harti. Optimal foraging in glossophagine bats: when to give up. *Am. Nat.*, 115(5):696–704, 1980.

[23] J. Gibbon and R. M. Church. Representation of time. *Cognition*, 37:23–54, 1990.

[24] S. C. Hinton and W. H. Meck. How time flies: Functional and neural mechansims of interval timing. In C. M. Bradshaw and E. Szadabi, editors, *Time and Behaviour: Psychological and Neurobehavioural Analyses*. Elsevier Science, 1997.

[25] CR Gallistel and J. Gibbon. Time, rate and conditioning. *Psychological Review*, 107(2):289–344, 2000.

[26] Dani Brunner, Alex Kacelnik, and John Gibbon. Optimal foraging and timing processes in the starling, sturnus vulgaris: effect of inter-capture interval. *Animal Behaviour*, 44(4):597 – 613, 1992.

[27] Richard S. Sutton. Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *In Proceedings of the Seventh International Conference on Machine Learning*, pages 216–224. Morgan Kaufmann, 1990.

[28] M. Anderson. A flexible approach to quantifying various dimensions of environmental complexity. In E.R. Messina and A.M. Meystel, editors, *Research and Engineering of Intelligent Systems*, 2004.

[29] Michael L. Anderson, Tim Oates, Waiyian Chong, and Donald Perlis. The metacognitive loop i: Enhancing reinforcement learning with metacognitive monitoring and control for improved perturbation tolerance. *J. Exp. Theor. Artif. Intell.*, 18(3):387–411, 2006.

[30] Angela J. Yu and Peter Dayan. Expected and unexpected uncertainty: Ach and ne in the neocortex. In *NIPS*, pages 157–164, 2002.

[31] Angela J. Yu and Peter Dayan. Uncertainty, neuromodulation, and attention. *Neuron*, 46(4):681–692, May 2005.

[32] D.J. Bucci, P.C. Holland, and M. Gallagher. Removal of cholinergic input to rat posterior parietal cortex disrupts incremental processing of conditioned stimuli. *Journal of Neuroscience*, 18:8038–8046, 1998.

[33] V. Devauges and S.J. Sara. Activation of the noradrenergic system facilitates an attentional shift in the rat. *Behav. Brain Res.*, 39:19–28, 1990.

[34] Encyclopædia Britannica. Time perception. In *Encyclopædia Britannica*. 2009.

[35] Chong Liu and Jonathan Shapiro. Implementing classical conditioning with spiking neurons. In *ICANN (1)*, pages 400–410, 2007.

[36] R. A. Rescorla and A. R. Wagner. A theory of pavlovian conditioning: The effectiveness of reinforcement and non-reinforcement. In A H Black and W F Prokasy, editors, *Classical Conditioning II: Current Research and Theory*, pages 64–69. Aleton-Century-Crofts, 1972.

[37] R. S. Sutton and A. G. Barto. Time-derivative models of pavlovian conditioning. In M. Gabriel and J. W. Moore, editors, *Learning and Computational Neuroscience*, pages 497–537. MIT Press, 1990.

[38] R. S. Sutton. *Temporal Credit Assignment in Reinforcement Learning*. PhD thesis, University of Massachusetts, 1984.

[39] B. Porr and F. Wörgötter. Isotropic sequence order learning. *Neural Computation*, 15:831–864, 2003.

[40] F. Wörgötter. Actor-critic models of animal control — a critique of reinforcement learning. In *Proceeding of Fourth International ICSC Symposium on Engineering of Intelligent Systems*, 2004.

[41] F. Wörgötter and B. Porr. Temporal sequence learning, prediction, and control: A review of different models and their relation to biological mechanisms. *Neural Computation*, 17:245–319, 2005.

[42] D. E. Rumelhart and J. L. McClelland, editors. *Parallel Distributed Processing: Explorations in the Microstructures of Cognition. Volume 1: Foundations*. MIT Press, 1986.

[43] H. R. Berenji. Artificial neural networks and approximate reasoning for intelligent control in space. In *American Control Conference*, pages 1075–1080, 1991.

[44] A. W. Moore, C. G. Atkeson, and S. Schaal. Memory-based learning for control. Technical Report CMU-RI-TR-95-18, CMU Robotics Institute, 1995.

[45] Dimitri P. Bertsekas. *Dynamic programming: deterministic and stochastic models*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1987.

[46] Richard S. Sutton. Learning to predict by the methods of temporal differences. *Mach. Learn.*, 3(1):9–44, 1988.

[47] Herbert Robbins and Sutton Monro. A stochastic approximation method. *The Annals of Mathematical Statistics*, 22(3):400–407, 1951.

[48] C. J. C. H. Watkins and P. Dayan. Q-learning. *Machine Learning*, 8:279–292, 1992.

[49] Tommi Jaakkola, Michael I. Jordan, and Satinder P. Singh. Convergence of stochastic iterative dynamic programming algorithms. *Neural Computation*, 6:1185–1201, 1994.

[50] John N. Tsitsiklis and Richard Sutton. Asynchronous stochastic approximation and q-learning. In *Machine Learning*, pages 185–202, 1994.

[51] Satinder Singh, Tommi Jaakkola, Michael L. Littman, and Csaba Szepesv Ari. Convergence results for single-step on-policy reinforcement-learning algorithms. In *Machine Learning*, pages 287–308, 1998.

[52] Leemon Baird. Residual algorithms: Reinforcement learning with function approximation. In *In Proceedings of the Twelfth International Conference on Machine Learning*, pages 30–37. Morgan Kaufmann, 1995.

[53] P. Dayan and L. F. Abbott. *Theoretical Neuroscience: Computational and Mathematical Modeling of Neural Systems*. MIT Press, 2001.

[54] Ivan Petrovich Pavlov. *Conditioned reflexes*. Oxford University Press, Oxford, 1927.

[55] Bitterman ME, LoLordo VM, Overrnier JB, and Rashotte ME. *Animal learning*. Plenum Press, New York, 1979.

[56] Rescorla RA. *Pavlovian second-order conditioning: studies in associative learning.* Lawrence Erlbaum Associates, Hillsdale, N.J., 1980.

[57] Rainer Malaka. Models of classical conditioning. *Bulletin of Mathematical Biology*, pages 33–83, 1999.

[58] W. Schultz, P. Dayan, and P. R. Montague. A neural substrate of prediction and reward. *Science*, 275:1593–1599, 1997.

[59] W. Schultz. Predictive reward signal of dopamine neurons. *Journal of Neurophysiology*, 80:1–27, 1998.

[60] Herbert Robbins. Some aspects of the sequential design of experiments. *Bulletin of the American Mathematical Society*, 58:527–535, September 1952.

[61] Richard Bellman. A problem in the sequential design of experiments. *Sankhya: The Indian Journal of Statistics (1933-1960)*, 16(3/4):221–229, 1956.

[62] J. C. Gittins. Bandit processes and dynamic allocation indices. *Journal of the Royal Statistical Society. Series B (Methodological)*, 41(2):148–177, 1979.

[63] Donald A. Berry and Bert Fristedt. *Bandit Problems: Sequential Allocation of Experiments.* Chapman and Hall, 1985.

[64] Pierre-Simon Laplace. Mémoire sur la probabilité des causes par les évènemens. *Mem. Acad. Roy. Sci.*, 6:621Ð–656, 1774.

[65] Pierre-Simon Laplace. *Théorie Analytique des Probabilités.* Courcier Imprimeur, Paris, 3rd edition, 1812.

[66] E. T. Jaynes. *Probability Theory: The Logic of Science (Vol 1).* Cambridge University Press, 2003.

[67] Donald E. Knuth. *The Art of Computer Programming*, volume volume 2: Seminumerical Algorithms. Boston: Addison-Wesley, Boston, 3rd edition, 1998.

[68] Jeremy Wyatt. *Exploration and Inference in Learning from Reinforcement.* PhD thesis, University of Edinburgh, 1997.

[69] Leslie P. Kaelbling. *Learning in Embedded Systems*. The MIT Press, 1993.

[70] Steven J. Bradtke and Michael O. Duff. Reinforcement learning methods for continuous-time markov decision problems. In *Advances in Neural Information Processing Systems*, pages 393–400. MIT Press, 1994.

[71] Tapas Das, Abhijit Gosavi, Sridhar Mahadevan, and N. Marchalleck. Solving semi-markov decision problems using average reward reinforcement learning. *Management Science*, 45:560–574, 1999.

[72] Richard Sutton, Doina Precup, and Satinder Singh. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, 112:181–211, 1999.

[73] Martin Stolle and Doina Precup. Learning options in reinforcement learning. In *Lecture Notes in Computer Science*, pages 212–223, 2002.

[74] Satinder P. Singh, Tommi Jaakkola, Michael L. Littman, and Csaba Szepesvári. Convergence results for single-step on-policy reinforcement-learning algorithms. *Machine Learning*, 38(3):287–308, 2000.

[75] Samuel P. M. Choi, Dit yan Yeung, and Nevin L. Zhang. Hidden-mode markov decision processes for nonstationary sequential decision making. In *In Sequence Learning - Paradigms, Algorithms, and Applications*, pages 264–287. Springer-Verlag, 2001.

[76] K. Tsumori and S. Ozawa. Incremental learning in dynamic environments using neural network with long-term memory. In *Proceedings of the Int. Conf. on Neural Networks*, 2003.

[77] Richard S. Sutton. Generalization in reinforcement learning: Successful examples using sparse coarse coding. In *Advances in Neural Information Processing Systems 8*, pages 1038–1044. MIT Press, 1996.

[78] R. Duncan Luce. *Individual choice behavior*. John Wiley, Oxford, England, 1959.

[79] John S. Bridle. Training stochastic model recognition algorithms as networks can lead to maximum mutual information estimation of parameters. pages 211–217. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1990.

[80] Peter Dayan and Terrence J. Sejnowski. Exploration bonuses and dual control. *Mach. Learn.*, 25(1):5–22, 1996.

[81] T. Van der Zant, M. Wiering, and J. Van Eijck. On-line and real-time learning using the interval estimation algorithm. In *Proceedings of the 7th European Workshop on Reinforcement Learning*, pages 11–12, 2005.

[82] M. A. Wiering and Juergen Schmidhuber. Efficient model-based exploration. In R. Pfeiffer, B. Blumberg, J. A. Meyer, and S. W. Wilson, editors, *Proceedings of the Fifth International Conference on Simulation of Adaptive Behavior (SAB'98): From Animals to Animats*, volume 5, pages 223–228, 1998.

[83] Richard Dearden, Nir Friedman, and Stuart Russell. Bayesian q-learning. In *In AAAI/IAAI*, pages 761–768. AAAI Press, 1998.

[84] Ronald Howard. Information value theory. *IEEE Transactions on Systems Science and Cybernetics*, 2(1):22–26, 1966.

[85] Richard Dearden, Nir Friedman, and David Andre. Model based bayesian exploration. In *In Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence*, pages 150–159, 1999.

[86] Matthias Heger. Consideration of risk in reinforcement learning. In W. W. Cohen and H. Hirsh, editors, *Machine Learning: Proceedings of the Eleventh International Conference*, pages 105Ð–111. Morgan Kaufmann Publishers, 1994.

[87] Oliver Mihatsch and Ralph Neuneier. Risk-Sensitive reinforcement learning. *Mach. Learn.*, 49(2-3):267–290, 2002.

[88] Darsana P. Josyula, Michael L. Anderson, and Donald Perlis. Metacognition for dropping and reconsidering intentions. In *Metacognition in Computation*, pages 62–67, 2005.

[89] Michael L. Anderson and Donald R. Perlis. Logic, self-awareness and self-improvement: the metacognitive loop and the problem of brittleness. *J. Log. and Comput.*, 15(1):21–40, 2005.

[90] S. Dzeroski, L. De Raedt, and K. Driessens. Relational reinforcement learning. *Machine Learning*, 43:7–52, 2001.

[91] Leslie Pack Kaelbling, Tim Oates, Natalia Hernandez-Gardiol, and Sarah Finney. Learning in worlds with objects. In *The AAAI Spring Symposium*, 2001.

[92] Ir. M. van Otterlo. A survey of reinforcement learning in relational domains. Technical Report 2005-31, Department of Computer Science, University of Twente, 2005.

[93] Dimitri P. Bertsekas. *Dynamic Programming and Optimal Control*. Athena Scientific, 1995.

[94] Justin A. Boyan and Michael L. Littman. Exact solutions to time-dependent mdps. In *in Advances in Neural Information Processing Systems*, pages 1026–1032. MIT Press, 2000.

[95] M. A. Wiering. Reinforcement learning in dynamic environments using instantiated information. In *Proceedings of the Eighth International Conference on Machine Learning*, 2001.

[96] Bruno C. da Silva, Eduardo W. Basso, Filipo S. Perotto, Ana L. C. Bazzan, and Paulo M. Engel. Improving reinforcement learning with context detection. In *AAMAS '06: Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, pages 810–812, New York, NY, USA, 2006. ACM.

[97] Christian Balkenius and Jan Morén. Dynamics of a classical conditioning model. *Auton. Robots*, 7(1):41–56, 1999.

[98] R. P. N. Rao and T. J. Sejnowski. Spike-timing-dependent hebbian plasticity as temporal difference learning. *Neural Computation*, 13:2221–2237, 2001.

[99] S. J. Thorpe and M. Imbert. Biological constraints on connectionist models. In R. Pfeifer, Z. Schreter, F. Fogelman-Soulié, and L. Steels, editors, *Connectionism in Perspective*, pages 63–92. Elsevier, 1989.

[100] P. H. Jen, X. D. Sun, and P. J. Lin. Frequency and space representation in the primary auditory cortex of the frequency modulating bat Eptesicus fuscus. *Journal of Comparative Physiology*, 165(1):1–14, 1989.

[101] F. Rieke, D. Warland, R. de Ruyter van Steveninck, and W. Bialek. *Spikes — Exploring the Neural Code*. MIT Press, 1996.

[102] W. Gerstner and W. M. Kistler. *Spiking Neuron Models*. Cambridge University Press, 2002.

[103] M. T. Hagan, H. B. Demuth, and M. Beale. *Neural Network Design*. PWS Publishing Company, 1996.

[104] E. D. Adrian. The impulses produced by sensory nerve endings. *The Journal of Physiology*, 61:49–72, 1926.

[105] E. D. Adrian. *The Basis of Sensation: The Action of the Sense Organs*. W. W. Norton, 1928.

[106] S. J. Thorpe, D. Fize, and C. Marlot. Speed of processing in the human visual system. *Nature*, 381:520–522, 1996.

[107] S. J. Thorpe and J. Gautrais. Rank order coding. In J. Bower, editor, *Computational Neuroscience: Trends in Research 1998*, pages 113–119. Plenum Press, 1998.

[108] S. J. Thorpe, A. Delorme, and R. VanRullen. Spike-based strategies for rapid processing. *Neural Networks*, 14(6-7):715–726, 2001.

[109] R. VanRullen and S. J. Thorpe. Rate coding versus temporal order coding: What the retinal ganglion cells tell the visual cortex. *Neural Computation*, 13:1255–1283, 2001.

[110] V. B. Mountcastle. Modality and topographic properties of single neurons of cat's somatosensory cortex. *Journal of Neurophysiology*, 20:408–434, 1957.

[111] D. H. Hubel and T. N. Wiesel. Receptive fields, binocular interaction and functional architecture in the cat's visual cortex. *The Journal of Physiology*, 160:106–154, 1962.

[112] E. C. Kandel and J. H. Schwartz. *Principles of Neural Science.* Elsevier, 3rd edition, 1991.

[113] K. Doya. What are the computations of the cerebellum, the basal ganglia and the cerebral cortex. *Neural Networks*, 12:961–974, 1999.

[114] K. Doya. Complementary roles of basal ganglia and cerebellum in learning and motor control. *Current Opinion in Neurobiology*, 10:732–739, 2000.

[115] J. A. Anderson. A simple neural network generating interactive memory. *Mathematical Biosciences*, 14:197–220, 1972.

[116] T. Kohonen. Correlation matrix memories. *IEEE Transactions on Computers*, 21:353–359, 1972.

[117] T. Kohonen. *Self-organization and Associative Memory.* Springer-Verlag, 3rd edition, 1989.

[118] S. Grossberg. *Studies of Mind and Brain.* D. Reidel Publishing, 1982.

[119] W. B. Levy and O. Steward. Temporal contiguity requirements for longterm associative potentiation/depression in the hippocampus. *Journal of Neuroscience*, 8:791–797, 1983.

[120] B. Gustafsson, H. Wigstrom, W. C. Abraham, and Y.-Y. Huang. Long-term potentiation in the hippocampus using depolarizing current pulses as the conditioning stimulus to single volley synaptic potentials. *Journal of Neuroscience*, 7:774–780, 1987.

[121] D. Debanne, B. T. Gahwiler, and S. H. Thompson. Asynchronous pre- and postsynaptic activity induces associative long-term depression in area CAI of the rat hippocampus in vitro. In *The Proceedings of the National Academy of Sciences (USA)*, volume 91, pages 1148–1152, 1994.

[122] H. Markram, J. Lübke, M. Frotscher, and B. Sakmann. Regulation of synaptic efficacy by coincidence of postsynaptic APs and EPSPs. *Science*, 275:213–215, 1997.

[123] J. C. Magee and D. Johnston. A synaptically controlled, associative signal for Hebbian plasticity in hippocampal neurons. *Science*, 275:209–213, 1997.

[124] C. C. Bell, V. Z. Han, Y. Sugawara, and K. Grankt. Synaptic plasticity in a cerebellum-like structure depends on temporal order. *Nature*, 387:278–281, 1997.

[125] G. Q. Bi and M. M. Poo. Synaptic modifications in cultured hippocampal neurons: Dependence on spike timing, synaptic strength, and postsynaptic cell type. *Journal of Neuroscience*, 18:10464–10472, 1998.

[126] D. Debanne, B. Gahwiler, and S. Thompson. Long-term synaptic plasticity between pairs of individual CA3 pyramidal cells in rat hippocampal slice cultures. *The Journal of Physiology*, 507:237–247, 1998.

[127] L. I. Zhang, H. W. Tao, C. E. Holt, W. A. Harris, and M.-M. Poo. A critical window for cooperation and competition among developing retinotectal synapses. *Nature*, 395:37–44, 1998.

[128] V. Egger, D. Feldmeyer, and B. Sakmann. Coincidence detection and changes of synaptic efficacy in spiny stellate neurons in rat barrel cortex. *Nature Neuroscience*, 2:1098–1105, 1999.

[129] D. Feldman. Timing-based LTP and LTD at vertical inputs to layer II/III pyramidal cells in rat barrel cortex. *Neuron*, 27:45–56, 2000.

[130] M. Nishiyama, K. Hong, K. Mikoshiba, M. Poo, and K. Kato. Calcium release frominternal stores regulates polarity and input specificity of synaptic modification. *Nature*, 408:584–588, 2000.

[131] V. Z. Han, K. Grant, and C. C. Bell. Reversible associative depression and nonassociative potentiation at a parallel fiber synapse. *Neuron*, 27:611–622, 2000.

[132] G.-Q. Bi and M. Poo. Synaptic modification by correlated activity: Hebb's postulate revisited. *Annual Review of Neuroscience*, 24:139–166, 2001.

[133] P. D. Roberts and C. C. Bell. Spike timing dependent synaptic plasticity in biological systems. *Biological Cybernetics*, 87:392–403, 2002.

[134] R. Kempter, W. Gerstner, and J. L. van Hemmen. Hebbian learning and spiking neurons. *Phys. Rev. E*, 59(4):4498–4514, 1999. article.

[135] P. J. Sjöström, E. A. Rancz, A. Roth, and M. Häusser. Dendritic excitability and synaptic plasticity. *Physiological reviews*, 88(2):769–840, April 2008.

[136] G. Chechik. Spike-timing-dependent plasticity and relevant mutual information maximization. *Neural Computation*, 15:1481–1510, 2003.

[137] A. J. Bell and L. C. Parra. Maximising information yields spike timing dependent plasticity. In *Proceedings of Advances in Neural Information Processing (NIPS)*, volume 18, 2004.

[138] T. Toyoizumi, J. P. Pfister, K. Aihara, and W. Gerstner. Spike-timing dependent plasticity and mutual information maximization for a spiking neuron model. In *Proceedings of Advances in Neural Information Processing (NIPS)*, volume 18, 2004.

[139] S. Haykin. *Neural network — A Comprehensive Foundation*. Prentice Hall, second edition, 1999.

[140] F. Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological Review*, 65:386–408, 1958.

[141] Marvin L. Minsky and Seymour A. Papert. *Perceptrons*. The MIT Press, December 1969.

[142] B. Widrow and M. E. Hoff. Adaptive switching circuits. *IRE WESCON Convention Record*, 4:96–104, 1960.

[143] B. Widrow and S. D. Stearns. *Adaptive Signal Processing*. Prentice-Hall, 1985.

[144] P. J. Werbos. *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*. PhD thesis, Harvard University, 1974.

[145] D. E. Rumelhart, G. E. Hintont, and R. J. Williams. Learning representations by back-propagating errors. *Nature*, 323:533–536, 1986.

[146] R. Legenstein, C. Naeger, and W. Maass. What can a neuron learn with spike-timing-dependent plasticity? *Neural Computation*, 17:2337–2382, 2005.

[147] D. Barber. Learning in spiking neural assemblies. In *Advances in Neural Information Processing Systems*, 15, pages 149–156. MIT Press, 2003.

[148] J. P. Pfister, D. Barber, and W. Gerstner. Optimal hebbian learning: A probabilistic point of view. *Lecture Notes in Computer Science*, 2714:92–98, 2003.

[149] R. E. Suri. A computational framework for cortical learning. *Biological Cybernetics*, 90:400–409, 2004.

[150] J. P. Pfister, T. Toyoizumi, D. Barber, and W. Gerstner. Optimal spike-timing-dependent plasticity for precise action potential firing in supervised learning. *Neural Computation*, 18:1318–1348, 2006.

[151] Barto AG. Adaptive critics and the basal ganglia. In Houk JC, Davis JL, and Beiser DG, editors, *Models of Information Processing in the Basal Ganglia*, pages 215–232. MIT Press, 1995.

[152] James C. Houk and S. P. Wise. Distributed modular architectures linking basal ganglia, cerebellum, and cerebral cortex: their role in planning and controlling action. *Cerebral Cortex*, 5:95–110, 1995.

[153] J. R. Wickens, A. J. Begg, and G. W. Arbuthnott. Dopamine reverses the depression of rat corticostriatal synapses which normally follows high-frequency stimulation of cortex in vitro. *Neuroscience*, 70(1):1–5, 1996.

[154] P. Calabresi, A. Pisani, N. B. Mercuri, and G. Bernardi. The corticostriatal projection: from synaptic plasticity to dysfunctions of the basal ganglia. *Trends in Neurosciences*, 19(1):19–24, 1996.

[155] Eugene M. Izhikevich and Niraj S. Desai. Relating stdp to bcm. *Neural Computation*, 15:1511–1523, 2003.

[156] John Hertz and Adam Prügel-Bennett. Learning short synfire chains by self-organization. *Network: Computatioon in Neural Systems*, 7(2):357–363, 1996.

[157] L. Perrinet and M. Samuelides. Coherence detection in a spiking neuron via hebbian learning. *Neurocomputing*, 44–46:133–139, 2002.

[158] S. M. Bohte, H. La Poutre, and J. N. Kok. Unsupervised clustering with spiking neurons by sparse temporal coding and multilayer rbf networks. *IEEE Transactions on Neural Networks*, 13:426–435, 2002.

[159] Xiaoli Tao and Howard E. Michel. Data clustering via spiking neural networks through spike timing-dependent plasticity. In *IC-AI*, pages 168–173, 2004.

[160] D.T. Pham, M.S. Packianather, and E.Y.A. Charles. A novel self- organised learning model with temporal coding for spiking neural networks. In *Intelligent Production Machines and Systems*, pages 307–312. Elsevier, 2006.

[161] Krzysztof J. Cios, Waldemar Swiercz, and William Jackson. Networks of spiking neurons in modeling and problem solving. *Neurocomputing*, pages 99–119, 2004.

[162] J. Nielsen and H. H. Lund. Spiking neural building block robot with hebbian learning. In *Proceedings of IROSÕ03. IEEE.* Press, 2003.

[163] S. Song, K. D. Miller, and L. F. Abbott. Competitive Hebbian learning through spike-timing-dependent synaptic plasticity. *Nature neuroscience*, 3(9):919–926, 2000.

[164] Andrew Carnell. An analysis of the use of hebbian and anti-hebbian spike time dependent plasticity learning functions within the context of recurrent spiking neural networks. *Neurocomputing*, 72:685–692, 2009.

[165] Stephen G. Eick. The two-armed bandit with delayed responses. *The Annals of Statistics*, 16(1):254–264, 1988.

[166] András György, Levente Kocsis, Ivett Szabó, and Csaba Szepesvári. Continuous time associative bandit problems. In Manuela M. Veloso, editor, *IJCAI*, pages 830–835, 2007.

[167] Donald E. Knuth. *The Art of Computer Programming*, volume volume 1: Fundamental Algorithms. Boston: Addison-Wesley, Boston, 3rd edition, 1998.

[168] Mikel J. Harry. *The Nature of six sigma quality.* Motorola University Press, 1988.

[169] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition*. The MIT Press, 2001.

[170] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, December 1959.