Multiple Model Reinforcement Learning for Environments with Poissonian Time Delays

by

Jeffrey Scott Campbell

A Thesis submitted to

the Faculty of Graduate Studies and Research

in partial fulfilment of

the requirements for the degree of

Master of Applied Science

in

Electrical Engineering

Carleton University

Ottawa, Ontario, Canada

April 2014

Abstract

This thesis proposes a novel algorithm for use in reinforcement learning problems where a stochastic time delay is present in an agent's reinforcement signal. In these problems, the agent does not necessarily receive a reinforcement immediately after the action that caused it. We relax previous constraints in the literature by assuming that rewards may arrive to the agent out of order or may even overlap with one another. The algorithm combines Q-learning and hypothesis testing to enable the agent to learn about the delay itself. A proof of convergence is provided. The algorithm is tested in a grid-world simulator in MATLAB, the Webots mobile-robot simulator, and in an experiment with a real e-Puck mobile robot. In each of these test beds, the algorithm is compared to Watkins' Q-learning, of which it is an extension. In all cases, the novel algorithm outperforms Q-learning in situations where reinforcements are variably delayed.

Table of Contents

Abstract ii

Table of Contents iii

List of Tables vi

List of Figures vii

List of Symbols ix

1 Introduction 1

1.1 Thesis Statement 4

1.2 Motivation 4

1.3 Contributions 5

1.4 Organization 6

2 Background 7

2.1 What is Reinforcement Learning? 7

2.1.1 RL compared to other forms of learning 10

2.1.2 How does Reinforcement Learning Work? 12

2.1.3 Methods of Reinforcement Learning 13

2.1.4 Generalizing in large environments 20

2.2 Challenges of Robotics Applications 22

2.2.1 Simulated-Policy Transfer 22
2.2.2 Reward Function Design 23
2.3 Delays 24
2.3.1 Constant delay 24
2.3.2 Variable delay 26

3 Problem Formulation 28
3.1 The Problem 28
3.1.1 Reinforcement Learning 29
3.1.2 Introducing Variable Delays 31
3.2 Multiple Model Q-Learning 34

4 Theoretical Results 43
4.1 Convergence 43

5 Simulation and Experimental Results 56
5.1 Grid world simulations 56
5.1.1 Multiple Model Q-Learning in the Grid World 65
5.2 Webots simulation - line-following robot 80
5.2.1 Problem Overview 81
5.2.2 State and Action Spaces 83
5.2.3 Comparing adaptive and traditional techniques 85
5.2.4 Comparing different learning techniques 87
5.3 Experimental Results 94
5.4 Discussion 100

6 Conclusion and Recommendations 102
6.1 Conclusion 102
6.2 Contributions 103

6.3 Recommendations for Future Work 105

List of References **108**

List of Tables

2.1 Simple Q-table - Rain or Shine 14

2.2 Optimistic Q-values 19

2.3 True Q-values 19

3.1 Roulette delay example 35

3.2 Initial Roulette Q-table 41

3.3 Roulette Q-table after round 1 41

3.4 Roulette Q-table after round 2 42

3.5 Roulette Q-table after round 3 42

5.1 Multiple model Q-values along the optimal route after 10^6 iterations . 72

5.2 e-Puck Action Space 84

List of Figures

2.1 The general reinforcement learning model 9

2.2 The general reinforcement learning model with reward delay 25

3.1 A Roulette table 32

3.2 Delay distribution when $\lambda = 5$ 33

3.3 Activation of a candidate model in multiple model Q-Learning 36

5.1 The 3 x 3 grid world 58

5.2 3 x 3 grid world - Effect of increasing the mean delay 60

5.3 Normalized average reward in various environment sizes for $\lambda = 5$ . . 61

5.4 Normalized average reward in various environment sizes for $\lambda = 50$ . . 62

5.5 Normalized average reward in various environment sizes for $\lambda = 150$. 63

5.6 9 x 9 grid world - Performance after 150k iterations for errors in the delay estimate $i = \hat{\lambda} - \lambda$ 66

5.7 The 9 x 9 grid world 67

5.8 The 9 x 9 grid world - state numbering 68

5.9 Relative performance of multiple model Q-learning vs single model Q-learning in the 9×9 grid world with random delays present 70

5.10 How often each state is visited during learning when using multiple model Q-learning. 74

5.11 Deterministic policy formed using multiple model Q-learning 75

5.12 Probabilistic policy formed using multiple model Q-learning 76

5.13 How often each state visited during learning when using single model Q-learning 77
5.14 Deterministic policy formed from using single model Q-learning . . . 78
5.15 Probabilistic policy formed from using single model Q-learning 79
5.16 e-Puck Mobile Robot 81
5.17 e-Puck Camera View 81
5.18 An Example Racetrack 82
5.19 Webots - Controllers' Performance Under Delay vs. Undelayed PID . 88
5.20 Webots - (single run) average reward during learning ($\lambda = 90$) 90
5.21 Webots - (single run) accumulated error during learning ($\lambda = 90$) . . 91
5.22 Webots - (5 runs) average reward during learning ($\lambda = 90$) 92
5.23 Webots - (5 runs) accumulated error during learning ($\lambda = 90$) 93
5.24 Comparison of Average Reward During Experiment 1 96
5.25 Comparison of Accumulated Error During Experiment 1 97
5.26 Comparison of Average Reward During Experiment 2 98
5.27 Comparison of Accumulated Error During Experiment 2 99

List of Symbols

Symbols	Definition
t	time-step index
s	current state
s'	next state
S	state set
a	current action
a^*	most valuable action
A	action set
s_d	state observation delay
a_d	action delay
r_d	reward delay
d	delay at time step t
λ	mean time delay
$\hat{\lambda}$	estimated value of λ

$\hat{\lambda}^*$	most valuable delay estimate
Λ	set of considered delay estimates
$P(s, a, s')$	probability of state transition from s to s' after performing a
$R(s, a, s')$	reward set for performing a in s and arriving in s'
γ	discount factor
α	stepsize parameter (learning rate)
π	control policy which maps $S \rightarrow A$
π^*	optimal control policy
$V(s)$	value of being in state s
$Q(s, a)$	value of being in s and performing a
$Q(s, a, \hat{\lambda})$	value of being in s and performing a, assuming that the mean delay is $\hat{\lambda}$
$V^*(s)$	true value of being in state s
$Q^*(s, a)$	true value of being in s and performing a
$Q^*(s, a, \hat{\lambda})$	true value of being in s and performing a, assuming that the mean delay is $\hat{\lambda}$
$e(s)$	eligibility of s
ϵ	chance of random exploration
o	number of time steps that state observations are delayed

a	number of time steps that state actions are delayed
k	number of time steps that state reinforcements are delayed
$F(t)$	history of the algorithm until $Q(t+1)$ is about to be computed
$E[z]$	expected value of z
$x(t)$	vector at time t
$x_i(t)$	i^{th} component of x
x^*	fixed point of $x(t)$
T^i	set of indices at which x_i is updated
$\tau_j^i(t)$	set of indices where updates occurred at or before the present $(0 \leq \tau_j^i(t) \leq t)$
$w_i(t)$	noise term
$F(x)$	mapping from $\mathbb{R}^n$ into itself
$(\Omega, \mathcal{F}, P)$	probability space

Chapter 1

Introduction

The human brain is the most adaptive and capable learning machine ever found. From the time we are born, each of us is bombarded with new situations and experiences. Somehow, our brains can control our behaviour so that we survive in a complex world by classifying stimuli, detecting patterns, and learning from the world around us. The dream of artificial intelligence is to build a machine which is as adept at dealing with a changing world as we humans. To this end, we can draw inspiration from the brains and nervous systems of humans and other life all around us.

Robots and other agents often employ manually generated kinematic or dynamic physical models which define the robots' bodies and parts of the external world [1]. Before we enter a new situation, we do not conjure up a mathematical model to predict what will happen to us. Instead, we produce a mental model of the situation on the fly. This is vital because new situations do not always warn us before they present themselves. We are interested in constructing machines which can generate their own models using information which they extract from the environment as they explore.

Reinforcement learning is one method to create internal models in novel situations. It is clear that, for our brains, at least some of our learning takes the form of reinforcement learning. That is, the brain can learn which behaviours are useful or inappropriate based on feedback from the environment. This feedback may be in the form of a stimulus which causes pleasure or pain; hunger or satiety. There are many feelings, emotions and drives which serve to reinforce different behaviours in life. Designers have drawn inspiration from this biological scheme in order to create a similar form of learning for our machines which we call machine reinforcement learning.

Machine reinforcement learning (RL) is a form of unsupervised machine learning. Unlike in forms of supervised learning, an unsupervised agent need not receive correct input/output pairs to facilitate its adaptation. Thus, to an extent, RL allows the agent to design its own behaviour without supervision. This property is especially useful when an agent's desired behaviour is difficult to explicitly engineer [2]. Instead, RL permits a designer to specify the properties of a behaviour (such as speed or smoothness) rather than explicitly defining the behaviour itself. In this way, the agent is given the capacity to explore its environment and invent emergent behaviours which fulfill the designer's constraints but which may not have been obvious [3]. It is also possible to embed a priori knowledge in the machine's initialization to provide a head start to the learning [4].

In most RL machines, it is assumed that the environment provides reinforcement immediately after a behaviour. This is to say that there is no time delay between an action and its respective reward. The agent acts, is rewarded or punished, and learns. Repeat. This assumption is fine in many applications. In particular, Q-learning is one RL technique which is used for myriad purposes, such as: multi-robot domains [5]; human-robot collaboration [6]; structure facade parsing [7]; robot navigation [8]; service discovery for wireless ad-hoc networks [9]; network traffic signal control [10]; nonplayer character decision making in video games [11]; computational modelling of electricity markets [12]; and dynamic pricing in electronic retail markets [13]. In these applications it is commonly assumed that the agent receives reinforcements (herein called rewards) instantaneously.

In some applications, this may not be the case. Instead, it is possible that the rewards be delayed in time. Such a delay has many possible causes. It may be due to unpredictable network latency as in an ad-hoc network or perhaps available sensors are of a poor quality. Also, in a swarm of robots, the reinforcements might be calculated in distributed fashion using the information from a subset of the full group. No matter the cause, problems of variable delay seem likely to appear more frequently as further work is done on decentralized control methods and swarm robotics. Consider the delay possibilities, for example, in ad-hoc networks between members of a flying unmanned aerial vehicle swarm [14]. Depending on which members are included in the calculation, the result may be variably delayed. We focus on the application of mobile robotics because delay problems of this kind could arise due to the complexity of the field. However, variable delay problems exist elsewhere as well.

If reinforcement signals are delayed by a constant amount, the problem can be dealt with because we can still be certain that if, for example, the delay is five time steps, the reinforcement was caused by the behaviour five time steps in the past. In this case, we need to remember more of our past actions than in an undelayed case.

When we consider our own lives, we recall situations when feedback was variably delayed in time. We may try a series of new things in response to a new situation. Later, we receive a positive result. We are not sure which of our actions actually caused the desirable outcome because the time delay is not constant. It is more difficult for us to assign responsibility to our actions the more variable the time delay. Although we seem to learn better when there is immediate or even constantly delayed feedback, we are still able to learn, albeit slower, when there is a variable delay.

The question of this thesis is: how can a machine learn in the presence of variable reinforcement time delays?

Superstition may be thought of as a side effect of delayed feedback which demonstrates that our learning can be partially inhibited by variable time delays. It also provides hints at how we might design a machine for this problem. Consider a baseball player who wins ten games in a row. Before each game, the player slept well, trained vigorously, and wore her lucky shoes. Many people will conclude that the shoes are indeed graced by Lady Luck, and continue to wear them for future baseball games, just in case.

The wide existence of superstitions in human culture suggests that when the results of our actions are unclear, our method of assigning blame is prone to error if we do not later verify these superstitions empirically. Intuitively, perhaps any machine learning method which deals with variable time delays should be able to form superstitions as we do, if only just to be able to rule them out later. This is the foundation of the novel multiple hypothesis testing method proposed herein.

1.1 Thesis Statement

Using multiple-model Q-learning as presented in this thesis, an agent can adapt to an environment with a variable time-delayed reinforcement signal by learning about the statistics of the delay. We assume that the time delay is Poissonian.

1.2 Motivation

This thesis will consider a situation where a machine reinforcement learning agent suffers from Poissonian time delays in its reinforcement signal. That is, all reinforcements will be delayed in time by a Poissonian random variable.

Variable delays of this kind have been considered in the literature, but there has been an assumption that the reinforcements can neither overlap nor arrive to the agent out of order [15]. In this thesis, that assumption is removed.

As a potential solution, this thesis will propose using multiple models of Q-learning in parallel to allow a learning agent to create superstitious estimates of the mean time delay and then test these hypotheses over time to eventually learn about the time delay itself so that it can be incorporated to reduce errors in credit assignment.

Such a method could find use in applications which involve heavy network latency (jitter), as this may introduce variable time delays [15]. In applications where control is decentralized, like in mobile robot swarms, reinforcement signals may depend on other robots in the swarm and thus there could be a delay for any one robot to receive its reinforcement signal.

1.3 Contributions

The contributions of this thesis are:

- a novel method of reinforcement learning to better handle Poissonian reinforcement delays;
- a proof of convergence for the novel method based on that of Q-learning as proposed in [16].
- in support of these:
  - a demonstration and exploration of aspects of the problem in a simulated grid-world environment,
  - line-following simulations of the novel method to observe its behaviour during learning, explore its convergence and demonstrate performance improvement relative to a simulated Q-learning agent;
  - real-world line-following experiments implementing the novel method to further demonstrate its improved performance relative to a Q-learning implementation; and

1.4 Organization

Chapter 2 will briefly survey reinforcement learning techniques and methods from the literature for dealing with variable time delays in an agent's reinforcement signal. Chapter 3 will present the notation and definitions which will be used to describe the problem throughout the thesis. The chapter will contain the formal presentation of the problem of a Poissonian-delayed reinforcement signal in the context of a reinforcement learning scheme and describe the problem in the general sense before presenting the novel method itself. Chapter 4 will contain the theoretical results of this thesis. Chapter 5 will express the problem in the specific case of a line-following mobile robot. This chapter will contain grid world simulation results, as well as simulation and experimental results pertaining to the line-following application. Finally, Chapter 6 will provide concluding remarks, summarize the contributions of this thesis, and propose related areas for future work.

Chapter 2

Background

This chapter contains a brief summary of reinforcement learning techniques in the literature, including methods for dealing with time delays in an agent's reinforcement signal.

2.1 What is Reinforcement Learning?

Reinforcement learning (RL) encompasses the set of optimization problems where an agent must learn to behave appropriately through trial-and-error in an incompletely-known environment [17]. Approaches to solve these problems typically have the agent explore a space of possible strategies and receive feedback based on the decisions it makes. The agent then uses this feedback to create a useful policy, or perhaps even an optimal policy [2,3]. RL techniques are based on how animals (including humans) learn, which means behaving in such a way as to seek rewards and avoid punishments.

For example, consider a man playing Chess. With reference to Figure 2.1, the man is the agent and the Chess board is the environment. The arrangement of the pieces on the board represents the state of the game. Each turn, the man considers the state of the game and decides which move to make. In RL, his move would be called an action. If he is to win, the man must generate a strategy to map game states to useful actions so that he can capture enemy pieces, and eventually, the enemy king. The mapping of states to actions is called a policy in RL. After each action, the game proceeds to a new state because the pieces have moved. Some moves are useful and others are not. If his action was a good one, he may capture an opponent's piece. If not, the action may have led him into his opponent's trap. Over time, the man is able to learn to improve at the game of chess by trying different strategies and observing the outcomes of the games and the individual plays. This observation of the results of actions is what we call feedback. For example, if the game proceeds to a point where the man has checkmated his opponent, then the feedback from this state is very valuable because the game is won. On the other hand, a game state that causes the man to lose his queen is a very negative event. In Chess, an RL agent would aim to learn how to win by trying different actions and strategies. The agent would keep track of the feedback it received while performing different strategies, perhaps in a table or some other representation. Over time, the RL agent would use what it has learned in the past to play better and better, until eventually, it has learned intelligent moves for many situations.

RL is closely tied to other domains, like the theory of optimal control, dynamic programming, stochastic programming, and optimal stopping. Whereas optimal control seeks to optimize a cost function based on a precise model of the system to be controlled, RL differs in that it does not assume to have perfect knowledge of a model for the system. This is an advantage in applications where it is difficult to produce a precise model or approximations must be made. RL can function even if it only has data received from its interactions with the environment [2].

RL is related to dynamic programming because both seek to divide and conquer complex problems. RL techniques define problems in terms of state and action spaces, as in the Chess example. These notions allow the problem of how to learn about the environment to be broken down into small steps which are taken many times. For

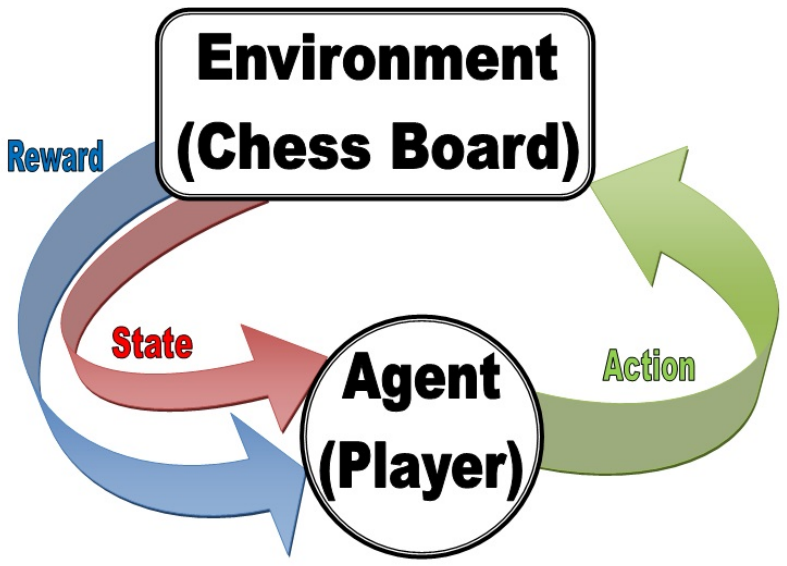


Figure 2.1: The general reinforcement learning model

example, Chess is a very complex game, but it can be simplified by considering one move at a time and making assumptions about all of the moves thereafter. This allows the RL agent to solve a simple problem of what to do immediately, as in dynamic programming, rather than having to brute-force a full-game strategy before the match has begun.

RL techniques attempt to estimate the value of performing an action in some given state. For example, an RL agent would try to assess the value of taking the queen, given that the board is in a particular state. RL is related to stochastic programming because these valuations are estimates. Estimates should be considered stochastic because they may be based on future uncertainty. Even if the man captures a knight in Chess while the game is in a particular state, different opponents may respond differently to his action. The reward for having taken that action is therefore stochastic in nature.

In a way, RL agents are trying to solve optimal stopping problems. The theory of optimal stopping addresses the problem of when to perform some action to maximize future expected reward or minimize future expected cost. This optimization is most readily thought of financially. Business must make decisions which maximize future earnings and minimize future costs.

There are two main groups of algorithms which seek to solve RL problems. The first group involves exploring a set of behaviours to find one which is well-suited to the agent's environment. Some examples of these techniques are genetic algorithms, genetic programming, simulated annealing, and multi-agent neural nets [18–20]. The second group exploits statistical techniques and dynamic programming (DP) to incrementally estimate the value of certain actions when the agent finds itself in various states in its environment. The novel algorithm proposed in this thesis is of the second variety.

In either case, the algorithm must have some measure of performance that it can learn from. For example, in the genetic algorithm, solution candidates receive a fitness score. Similarly, in dynamic programming techniques, there is usually a reward or punishment signal (called a reinforcement signal) which gives information about the quality or usefulness of a relevant action or outcome. Note that in these two examples, the fitness scores or rewards can be used to compare the solutions to one another, but they do not provide any information about the optimal solution. The agent can only know if it has improved or not, but it does not know the perfect answer. This means that the designer does not necessarily need to know what the perfect answer is either.

2.1.1 RL compared to other forms of learning

It is useful to understand how RL differs from other forms of machine learning. In supervised learning methods, the designer must present data to demonstrate correct choices in different situations [2]. These may be correct input/output pairs, such as in an artificial neural network, or even a demonstration of a useful strategy, such as in learning from demonstration [21]. In many of these probably-approximately-correct

(PAC) approaches there is a period of training where the agent may make mistakes as it discerns patterns from its data and then there is a period of testing where its performance is evaluated against an independent data set to gauge how well the agent has generalized [22]. The PAC framework is so named because the agent seeks to do as well as possible during testing, but generalization is rarely perfect.

RL methods are unsupervised methods which means that they do not require correct input/output pairs during learning. In other words, when the agent performs an action it is not told what the 'best' action would have been. RL actions are evaluated by a reinforcement signal, but nothing more. An unsupervised method is so named because the agent can be thought of as having no supervision. There is no supervisor to tell the agent what it should have done. Instead, the agent must look at the result of its behaviour. In contrast, in a supervised method, the agent has a kind of supervisor to tell it what the optimal action should have been in every situation. Thus, the supervised agent can measure how right or wrong it was compared to the optimal, while the unsupervised agent can only measure its performance relative to its own past performance. As a result, the agent must be able to learn to find the best actions from the feedback it receives from its many suboptimal actions during learning. Because unsupervised agents cannot compare to the optimal, they must continuously explore in hopes of reaching an optimal policy or in some cases, a PAC policy. In the latter case, we are satisfied with a policy which is ϵ-close to optimal with probability $1-\delta$ [23]. These parameters are necessary to specify because without knowledge of the optimal, it is possible for an unsupervised agent to become convinced that it has found the optimal only because it is the best policy so far, not because it is truly optimal.

2.1.2 How does Reinforcement Learning Work?

In RL, an agent's goal is to maximize the reward it receives over some period of time. In classical RL, problems are often modelled as Markov Decision Processes (MDPs). These are defined by a state set S and an action set A. The decision maker can perform an action $a \in A$ to move from state $s \in S$ to state $s' \in S$ with probability $P(s, a, s')$. Finally, actions in A can incur rewards from the reward set $R(s, a, s')$. We can express these MDPs as a 4-tuple $(S, A, P(s, a, s'), R(s, a, s'))$ [2, 3]. In the Chess example, the state-space S refers to the huge number of possible arrangements of pieces on the board. Any particular arrangement is the current state s. After a player makes a move a from the set of possible moves A, the state changes to s', a new arrangement which is still in the set S. The reward or value for making a move depends on how the board was arranged before the move, what the move was, and how the board looks afterwards, so the reward is a function of (s, a, s').

A finite MDP can be described by a 5-tuple (S, A, P, R, γ) where γ is an added discount factor. The discount factor is introduced so that immediate rewards are some amount more valuable than future rewards in the same way that money today is preferable to an equal amount of money tomorrow. A discount factor closer to one makes the agent far-sighted, while a discount factor close to zero makes the agent short-sighted. In this way, the discount factor can be used to give the agent some personality.

The state transitions in an MDP can be said to have the Markov property, which means that only the current state s is needed to inform the policy. In Chess, this can be interpreted to mean that our strategy at any point of the game depends only on the current arrangement of the board. Past arrangements are not important. When we introduce reward delays into the system, knowledge of the current state s is no longer sufficient because we cannot be certain as to which past state caused the rewards. If a dog receives a treat five minutes after it performs the corresponding trick, the dog must be able to remember up to five minutes in the past to be able to properly learn from the reward. If not, the credit for the treat will be assigned to something else which was not the real cause.

2.1.3 Methods of Reinforcement Learning

Value functions are one set of RL methods. This approach uses the *Bellman Principle of Optimality.* Regardless of the agent's initial state s and action a, all further decisions must be optimal from the state s' which resulted from state-action pair (s, a). Value function approaches refer to the value of a state $V(s)$ or to the value of a state-action pair $Q(s, a)$. This can be thought of as the value of being in s or the value of being in s and performing action a, respectively. The notion is useful because it respects the Markov property, meaning that the agent can act while considering only its current state [2].

In the game of Chess, $V(s)$ would refer to the value of being in the game state s. For example, a state where we were about to take the opponent's queen would be a valuable one, but if we were about to lose our queen, the state would be undesirable. Either way, the value assumes that we are going to act optimally after being in such a position. If we know the value of all states of the game, we will always make moves that will place us in the most valuable following state s'. Similarly, $Q(s, a)$ would refer to the value of being in state s and making move a and then acting optimally thereafter. This is the value of choosing to take the queen or lose our queen when in a board arrangement where this is possible. After the action is taken, we only choose the most valuable actions from then on.

When the value of a state-action pair $Q(s, a)$ is known, the agent can easily select the most valuable action a^* from a set of actions which are possible in state s by finding the maximum value. It has been shown that if an agent always selects the most valuable action a^* it will lead to an optimal, deterministic policy [2]. This means that the agent will have maximized its future rewards.

In discrete problems Q is often represented as a lookup table. Consider the simple example in Table 2.1 where the possible states are sunny and rainy and the possible actions are drive or bike. In this example, if it is sunny, we will prefer to bike because $Q(Sunny, Bike) > Q(Sunny, Drive)$. If it is rainy outside, we will choose to drive because $Q(Rainy, Drive) > Q(Rainy, Bike)$. Once an accurate Q-table is generated, the agent can easily make decisions by comparing values based on the observed state.

Table 2.1: Simple Q-table - Rain or Shine

	Actions	
States	Drive	Bike
Sunny	2	5
Rainy	1	-3

In continuous spaces function approximation is used to avoid a dimensionally complex discrete space [2]. This means that instead of having a look-up table, as in Table 2.1, we would have a function which depends on the state and action values. In this thesis, we have chosen an application which can be discretized so that the focus can be on the novel learning algorithm rather than the choice of function approximator.

There are three main sets of RL algorithms which all aim to estimate the true value of the environment. The true value can be thought of as a point which we hope that the algorithm will converge to. We denote them as either $V^*(s)$ or $Q^*(s, a)$. The three algorithm sets are called dynamic programming-based optimal control approaches, rollout-based Monte Carlo methods, and temporal difference methods [2].

Dynamic programming methods need models of the transition probabilities $P(s, a, s')$ and the reward function in order to calculate the value function. These methods seek to solve many simpler subproblems of the main problem and then combine these results to form a model and solution for the main problem. Models may be learned either through policy iteration or value iteration and need not be provided beforehand.

To reach an optimal policy and learn the value function, the dynamic programming methods begin with an arbitrary policy, (1) evaluate the current policy, and then (2) improve it. These two steps are repeated until the value function converges. The value function then represents the true value of the states or state-action pairs and the agent can make decisions with confidence, assuming that the underlying environment has not changed during learning (e.g. that the Chess board and rules of Chess are stationary) [2]. An example of dynamic programming is an inefficient algorithm to calculate the n^{th} term of the Fibonacci sequence $F(n)$. Such an algorithm would recursively calculate $F(n)$ by calculating $F(n-1) + F(n-2)$, which would have to calculate the additional values of $F(n-3)$ and $F(n-4)$, and so on until reaching $F(1)$ and $F(0)$. The bigger problem of calculating $F(5)$ is thus broken down into several smaller problems.

Monte Carlo methods are different in that they estimate the value function by sampling the environment. An advantage of Monte Carlo techniques is that they do not require an explicit transition function. The current policy is tested on the environment and the rates of different transitions are recorded and then these are used in the estimation of the value function [2]. As the number of samples increases, the Monte Carlo method's model better approximates the environment.

Temporal difference methods sample the environment and employ differences between old estimates and new estimates of the value function, which is called the temporal error. An advantage of these approaches is that they need not wait until the end of an episode to have an available estimate of the agent's return, unlike in

Monte Carlo methods. Instead, they await only the following time step. The distinction is important because the ‘end’ of an ongoing task may be difficult to define [24]. Temporal difference methods combine ideas from Monte Carlo and dynamic programming methods in that they sample the environment according to a policy but also update estimates using past estimates. The novel algorithm proposed in this thesis is an extension of existing temporal difference methods.

Sutton introduced TD(λ) algorithms which are examples of temporal difference methods. The λ in this algorithm differs from the λ to be used in the remainder of this thesis. These algorithms iteratively estimate the values of states $V(s)$ as:

$$V_{k+1}(s) = V_k(s) + \alpha(r + \gamma V_k(s') - V_k(s))e(s) \tag{2.1}$$

where r is the reward, $e(s)$ is the eligibility of state s, $V(s)$ is the value of the current state, $V(s')$ is the value of the next state s', γ is a discount factor, $0 \leq \alpha \leq 1$ is the learning rate, and $0 \leq \lambda \leq 1$ is the trace decay parameter where:

$$e(s) = \sum_{k=1}^{t} (\lambda\gamma)^{t-k} \delta_{s,s_k}, \text{where } \delta_{s,s_k} = \begin{cases} 1 \text{ if } s = s_k \\ 0 \text{ otherwise} \end{cases}$$

In this way, states are assigned eligibility for credit for a reinforcement based on how frequently and recently they were visited in the past. $TD(0)$ is a special case where eligibility traces are not used. A state which has been visited very recently and frequently in the past will receive a higher eligibility. Consider the Chess example. If a player captures the queen, then the board arrangement (state) that allowed him to make the capture should receive the credit. However, the previous states that led to the current state are also important for the play. Instead of assigning credit for the reward to the current state alone, we can assign credit into the past as well.

If $\lambda = 0.9$ it means that we give full credit to the current state, 90% credit to the previous state, $\lambda^2 = 81\%$ credit to the state before that, and so on. States that were part of the 'team' that led to the reward are all eligible for credit. As a result, a sequence of historical states all become more valuable to the agent. In some cases, these eligibility traces can quicken convergence [25, 26].

Watkins' Q-learning is a natural extension of the $TD(0)$ algorithm [27, 28]. It can also be extended to include a form of eligibility traces, as in $TD(\lambda)$ [29]. Q-learning differs from $TD(\lambda)$ in that it seeks to evaluate state-actions pairs $Q(s, a)$ rather than just states themselves $V(s)$. The Q-learning algorithm updates its estimates as:

$$\begin{aligned} Q_{k+1}(s,a) = {} & Q_k(s,a) \\ & + \alpha[r + \gamma \max_a Q_k(s',a) - Q_k(s,a)] \end{aligned} \tag{2.2}$$

The form of Equation (2.2) is very similar to that of $TD(\lambda)$ in Equation (2.1). The main difference is that instead of using only the value of the next state $V(s')$, Q-learning uses the value of the best action for state s', that is, the maximum value for s'.

This thesis uses an expanded version of Q-learning to handle stochastically time-delayed rewards. To that end, we seek to expand Q-learning to include a delay-estimate dimension so that the algorithm can gauge the value of state-action-delay triplets. Q-learning is described further in detail in Chapter 3.

Because unsupervised algorithms are not presented with optimal answers, a major concern in RL problems is how an agent should balance exploration and exploitation. The more an agent explores its environment, the more likely it is to find the best sources of reinforcement. However, when an agent chooses to explore new behaviours, it must accept that it will not receive reinforcement for exploiting its best-so-far behaviours. Explore too much and many actions will be of low quality. Explore too little and many better possibilities will remain unknown. To maximize reinforcement in the long-term, the agent must have a smart procedure for balancing exploration and exploitation. There are several common techniques.

The first technique is a greedy strategy. In this strategy, the agent always selects the action which it expects to be the most valuable. A drawback of this method is that the agent is prone to becoming trapped in exploiting a suboptimal behaviour because it does not explore and never learns of the existence of superior behaviours.

A modification to the greedy strategy is to instil optimism in the agent. To do so, we initialize the agent with artificially high prior expectations about its environment. As a result, the agent is repeatedly disappointed by its choice of behaviour which causes it to explore and try new ones, optimistically expecting them to be better. The more optimistic the agent, the lower the chance of insufficient exploration. For example, let us say that the weather is sunny and the agent chooses to drive. The agent discovers that the value of doing this is 2. If we had initialized the Q-table to zero (not optimistic), and the agent acts greedily, then $Q(Sunny, Drive) > Q(Sunny, Bike)$ even though the true value of biking is higher for the sunny state. When it is sunny, the agent is now stuck in a sub-optimal behaviour of driving while sunny because it has never tried biking when it is sunny. Now, instead, consider an initialized Q-table as in Table 2.2. Even if we determine that driving while sunny has a value of 2, we still optimistically expect that biking will be better. If we act greedily, we will try biking because we have not tried it yet and are optimistic about it. We may find that it is better or worse, but at least now we have more information about the different actions. Now that both actions are explored, we can greedily choose the best action when it is sunny. This technique has been widely used in [30].

Another method to encourage exploration is called ϵ-Greedy exploration. In this strategy, the agent acts randomly with probability ϵ. Otherwise, it acts greedily.

Table 2.2: Optimistic Q-values

	Actions	
States	Drive	Bike
Sunny	10	10
Rainy	10	10

Table 2.3: True Q-values

	Actions	
States	Drive	Bike
Sunny	2	5
Rainy	1	-3

This may encourage us to try to bike for example, even if we currently think that it is better to drive. For example, we may set $\epsilon = 0.05$, which means that the agent will choose actions randomly 5% of the time. It is not necessary to keep ϵ constant. Sometimes, it can be beneficial to decrease epsilon as learning progresses. Intuitively, the agent should explore more when it is in an unfamiliar domain and explore less as it has learned more about the environment. This parallels the way that humans learn. Small children and teenagers are often much more curious (and reckless) than older people. We can tune the ϵ value to give some personality to the learning agent. Simulations and experiments in this thesis use optimistic exploration combined with ϵ-Greedy exploration.

There are more complicated and intelligent methods for exploring too. In addition to using the expectation of value represented in the Q-table, the agent may include additional statistics in its decision. For example, the agent may calculate a value-variance for its potential decisions and select its behaviour based on some confidence measure. In this way, it can explore in a way that increases its confidence about its expectations in the environment [17]. In terms of our example, if we have tried driving when it is raining many times, the variance of our estimate of its value will be relatively lower than that of biking if we have rarely biked while raining. We may force the agent to choose higher variance (lower certainty) actions to improve our estimates for those possibilities and encourage exploration. This technique allows the agent to minimize the uncertainty of its Q-table.

2.1.4 Generalizing in large environments

In more complex problems, it may become necessary to generalize the state-space when the resulting table would be too massive. In these cases, there are preferable methods to represent the environment instead of a table, as is often done in Q-learning or $TD(\lambda)$. For example, Tesauro used backpropagation as a function approximator for the value of the myriad states in backgammon [31]. There are many other function approximator techniques which might be used to estimate the value function when the number of states is very high or the problem is continuous. Decision trees [32] and variable resolution dynamic programming [33] are just two examples of such techniques. These techniques can permit the designer to abstract the state-space into something more manageable or intuitive. For example, in Chess, instead of representing every possible board combination as a separate state, we may group similar board arrangements into sets to reduce the number of possible states. We might also invent some function which measures the opportunity or risk of the current board arrangement and have that as a state in the problem. In this situation, the agent might learn to behave aggressively when the board is full of opportunity and conservatively when the board is full of risk without having to enumerate tremendous number of possible board combinations.

Similarly, it can also be useful to generalize about large action space. Some works in the literature employ neural networks to map action-sets to Q-values [34, 35]. The neural networks classify actions so that we need only map states to the smaller number of classes of actions, rather than the many actions themselves. In other work, some have used the Fourier basis for similar purposes [36]. Action space generalization can be thought of like abstraction in programming. Instead of working with individual assembly instructions, it can be perfectly suitable in many applications to consider C instructions, which are sets of many assembly instructions. In this way, the agent (or the programmer in programming) does not need to worry about as many small details or possibilities. Instead, the agent can consider a smaller number of sets of actions or even a function representing some continuous action, like a voltage level. In many sports, these sets of actions are called plays. It is easier for humans to learn about plays rather than learn all the actions within the plays. The drawback is that whenever an agent is given an action space that is a generalization, it must by definition exclude some specifics which may or may not be useful. For example, if the game of Chess is abstracted so that the state of the game is represented by classes of board arrangements, then some information about the actual board arrangement must be lost. We can only hope that the lost information was unimportant.

The choice of function approximator for generalization is important. Sutton recommended the use of sparse-coarse-coded function approximators (CMACs) [26] because other global function approximators were found to void the guarantee that value function updates would never increase error [37]. The careful selection of function approximator can guarantee convergence, but not always optimal convergence [38]. To avoid these concerns, we use a table representation in the simulations and experiments contained in this thesis so that the focus remains on the novel algorithm and not on the choice of function approximator.

If generalization is not appropriate, some large state spaces can be dealt with hierarchically. In this way, they are broken into several smaller learning problems to increase computational efficiency. Some techniques use a gating function to choose between available behaviours, each of which maps environment states into low-level actions [39, 40]. For example, we might implement a subsumption architecture into a Chess-playing agent to make a hierarchy of behaviours. At a lower level, the agent would avoid situations where its own valuable pieces are placed in immediate danger, while at a higher level, the agent might seek ways to trap its opponent, using the lower-level ability to avoid danger to itself.

Finally, as is done in this thesis, we may instead choose to discretize the space to make it more manageable, while still facilitating rapid learning and quality performance. In some applications, it may be necessary to specifically discretize by hand, as in [41, 42]. In others, the discretization may be created in the form of meta-actions which are intelligent combinations of more basic actions [43] or may be learned from data [44] itself. As an example, Chapter 5 details how the line-following problem used in this thesis is discretized.

2.2 Challenges of Robotics Applications

This thesis will use robotics applications to evaluate RL techniques. This is a particularly challenging application area for several reasons and, as a result, provides a robust test bed for algorithms. First, it can be difficult or expensive to provide real-world experience to a RL robot because of the need for repair, manual resets and slow execution speed of each episode. For example, a flying robot may routinely crash during learning and incur costly repair bills [2]. In the case of our line-following robot, manual resets are required during training when the robot loses sight of its track for too long.

2.2.1 Simulated-Policy Transfer

Simulations may be used instead, but because many robotic applications take place in continuous spaces, we must decide an appropriate resolution at which to discretize the problem. As well, simulation modelling errors may doom the simulation to succeed. This is to say that the simulation may be too simple or unrealistic in ways that are difficult to notice, but which allow the simulated robot to perform well when in reality, the robot will fail. To make matters worse, robotic systems are often cursed with high dimensionality and real-world samples can be fraught with noise and errors, which may make it more difficult for an agent to measure its state or accurately judge its rewards. Even if a robot is able to construct a useful policy, the policy may become inadequate over time if the dynamics of the robot change due to environmental factors or wear. The transfer of simulation-learned policies to real-world test beds can be problematic in applications where instability is more likely, such as in a pole-balancing robot [2]. These factors combine to make robotic applications robust test beds for algorithms.

2.2.2 Reward Function Design

It can also be problematic to design an appropriate reward function to encourage the desired behaviour, although this is usually simpler than explicitly programming a behaviour [2]. It is important to design a reward that leads to success but also defines elegance for the task. For example, we may want an agent to succeed at the game of Chess, but a binary win/lose reward function would provide sparse information for learning. Instead, we would like the agent to be able to learn about the most elegant ways to win. The agent can be imbued with a notion of elegance through well-designed reward functions (reward-shaping) [45] or through inverse reinforcement learning, a technique which crafts a reward function after examining expert demonstrations of the task to be learned [46]. For example, we might design a reward function for Chess based on the strategies of a master Chess player.

2.3 Delays

Real-world robotic systems can encounter delays. These delays can be present in their sensors or in the case of robotic swarms, the delay may results from communications between members of the swarm. When these delays interfere with the RL agent's learning, the problem can become difficult. In particular, variable time delays can make it difficult for the robot to decide which actions were responsible for the reinforcements which it has received. Learning will fail if noise and lag cause bad behaviours to be inadvertently rewarded and good behaviours to be punished. Time delays may be constant or variable and they are the focus of this thesis.

2.3.1 Constant delay

There is work in the literature which addresses the problem of constantly delayed reinforcements. It has been shown that such a problem can be reduced to an undelayed problem by expanding the state space of the MDP [15,47]. In other words, the agent must remember enough of its past actions to be able to assign responsibility despite the time delay. For example, assume that a dog performs tricks, but does not receive a treat until a time delay of five minutes has elapsed. Instead of storing only the dog's current state, we might remember the entire last five minutes of the dog's state. The additional memory would ensure that the treat can be properly assigned to the trick which earned it. With the state-space augmented in this way, the problem can be treated as an undelayed one.

In the general delay case, we can imagine a finite MDP where there is a state observation delay s_d, an action delay a_d, and a reward delay r_d [47]. Let us define this finite MDP as an 8-tuple $(S, A, P, R, s_d, a_d, r_d, \gamma)$ where $s_d, a_d, r_d \geq 0$ are integers which represent the number of time steps that the state observations, actions and rewards are delayed. In other words, we cannot observe which state we have moved

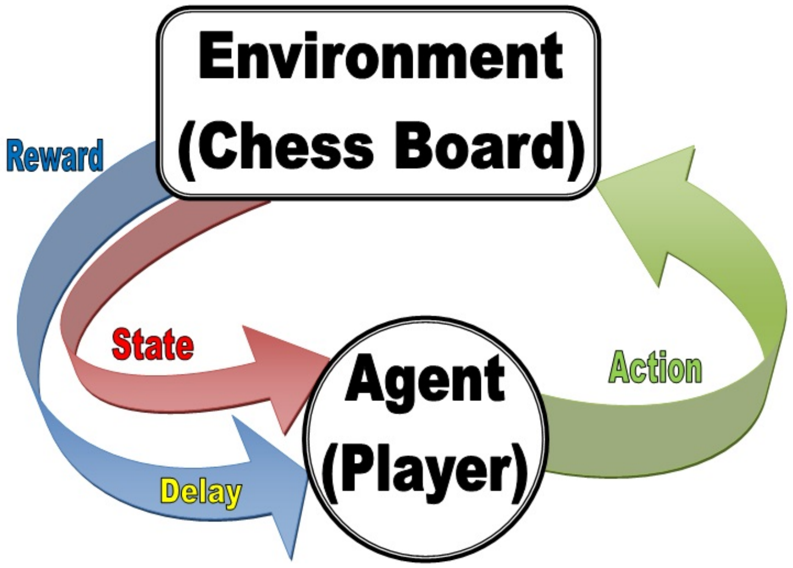


Figure 2.2: The general reinforcement learning model with reward delay

to for s_d time steps, we cannot perform the action a until a_d time-steps have elapsed, and all rewards are delayed r_d time steps after they were caused.

In this thesis, we will focus on the case where only a reward delay r_d is present. In the constant delay case, if we assume that $s_d = a_d = 0$ and $r_d \neq 0$, then we can model the problem as a constant-delayed MDP (CDMDP) which is defined by a 6-tuple $(S, A, P, R, \gamma, r_d)$ where $r_d \geq 0$ is an integer which is introduced to represent the number of time steps between when the agent was in state s and when it received reward r for having performed action a while in state s, r_d time steps ago. In the special case of undelayed RL, $r_d = 0$ means that rewards are received immediately after they are caused [47].

To deal with this constant delay, if r_d is known, we can assign credit for rewards into the past, rather than assigning credit to the last state to be visited. In this way, we shift the Markov property backwards in time to include the state at s_{t-r_d}. The result of this state-expansion is that the CDMDP can be reduced to an equivalent MDP [47].

2.3.2 Variable delay

There is also work in the literature dealing with variable time delay for states, actions, and rewards, but it comes with some assumptions which we will seek to relax. If it is assumed that the number of steps between successive reinforcements is a non-negative random variable, and the costs are collected in the proper order, then it is also possible to reduce this stochastic delayed MDP (SDMDP) to a standard undelayed MDP [15]. To reduce the problem in this fashion, it is necessary to assume that costs are collected in the proper order. For example, the reward from (s_1, a_1) will certainly be collected before the reward from (s_2, a_2). Because the order is respected, we can assign credit for the rewards with certainty. For example, we move through state-action pairs $(s_1, a_1), (s_2, a_2), (s_3, a_3)$ and eventually receive rewards $5, 3, 9$. We know that (s_1, a_1) caused the reward of 5, (s_2, a_2) caused the reward of 3, and (s_3, a_3) caused the reward of 9. A cost structure is made to keep track of the rewards which have been induced but not yet received, and when they are detected, they are assigned properly and removed from the cost structure.

In this thesis, we will relax the assumption that rewards arrive in the proper order, meaning that rewards are permitted to arrive in random order or even overlap with one another. In the case of overlapping, the rewards are summed and received as one combined reward. In the case where a time step contains no reward, the agent receives a reward of zero.

To extend the concept of a SDMDP, consider a *variably* delayed Markov decision process (VDMDP) which is characterized by (S, A, P, R, γ, d) where d is a random variable representing the number of time steps the reward is delayed away from its cause. This delay may be described by a stochastic distribution. For example, d may be Gaussian or Poissonian. For causality, assume that $d \geq 0$. Without loss of generality, in simulations and experiments in this thesis, we will assume that the delay is Poissonian. Finally, we assume that the agent will have no additional prior knowledge of the time delay other than a maximum cut-off, i.e. $d \leq d_{\max}$.

Because we do not assume that the rewards will necessarily arrive in the proper order, credit cannot be assigned properly with certainty. Instead, the agent must learn about the delay itself and seek to assign credit correctly with the highest probability attainable under the delay distribution.

Chapter 3

Problem Formulation

This chapter will describe the problem to be investigated and introduce the notation which will be used in the remainder of this thesis. Next, we will present the general solution and explain how it applies to the line-following robot application.

3.1 The Problem

We have a problem which can be modelled with a finite Markov Decision Process. We represent the MDP with a 5-tuple (S, A, P, R, γ) where S is the set of states within the environment, A is the set of actions the agent may choose, and P maps $S \times A \times S \mapsto [0, 1]$ which is the probability that taking action $a \in A$ while in state $s \in S$ will lead to state $s' \in S$. R is defined as the reward signal which maps $(S, A) \mapsto \mathbb{R}$ and γ is the discount factor to be applied to future rewards. Our goal is to allow an agent to develop a control policy which will perform well by some measure suitable to the application.

To illustrate these ideas, imagine a casino. There are many games at the casino which a gambler might choose to play. We can consider the gambler's current game to be the state. Each game has several actions the gambler can choose. Depending on the game and the action chosen, (s, a), he will receive a random reward. The average reward he can expect to receive from each action in each game differs. Over time, if the gambler had enough money to lose, it would be possible to find patterns and determine which game had the best payout, or lowest rate of loss in the case of a casino! Slot machines are a very simple example of a possible game, where the gambler merely pulls the handle and hopes for the best. On the other hand, a game like poker is more complex in terms of its state and action spaces. Fundamentally, the gambler is trying to generate a model of the casino, which, in general, is called the environment. If the casino can be explored fully and all games are thoroughly tested, then the gambler can create an optimal policy. In other words, the gambler can learn which games to play to maximize his expected profit. In this case, we hope that the gambler would learn to leave the casino!

3.1.1 Reinforcement Learning

Given that our goal is to model the environment accurately enough to be able to produce a control policy, we must select a technique to do so. One possibility is that we may use reinforcement learning (RL) to allow the agent to produce the control policy $\pi : S \mapsto A$ as it explores the environment. This policy maps the states to appropriate actions so as to maximize a value function, which is the expectation of future cumulative discounted rewards, given by:

$$V^{\pi}(s) = R(s) + \gamma \sum_{s'} P(s, \pi(s), s') V^{\pi}(s') \tag{3.1}$$

where $0 < \gamma < 1$. This can be thought of as the expected utility of being in state s and following policy $\pi(s)$ thereafter [47]. In the context of the casino, this equation can be interpreted to mean that the value of playing a particular game s is estimated as the immediate payout (reward) plus the estimated value of continuing to play that game afterwards, discounted by some factor γ. If the discount factor is set to zero, then the gambler does not consider the future, he only considers the immediate payout. If the discount factor is closer to one, then the future becomes more important to the gambler.

In some applications, performing an action a in state s moves the agent to a different state s'. In the casino example, playing one game does not somehow force the gambler to change to a different game. The gambler may play the same game indefinitely or switch games at will. In any case, the expected future value of a state depends on the probability of the agent actually reaching it. This is the meaning of the summation term.

In a casino, the reward scheme is already present in the rules of the games. However, in most applications, we must design the reward scheme in accordance with the goals of the application. For example, the rules of the casino are designed so as to make all gamblers lose money over time, but not too quickly, or else they would become frustrated and refuse to play. If a reward function focused on profit, the optimal policy would be to stay home. Instead, if a reward function focused on fun, the optimal policy would be different. In this way, the reward function is similar to the values of a human. Humans with different values behave differently from one another in the same situation. If we want an agent to behave a certain way, then we must imbue it with the proper values.

Now that we have defined the type of problem to be solved, we will use Q-learning to allow the agent to explore the environment and to learn over time. Q-learning is an example of an RL algorithm which could be used to solve the RL problem. In tabular Q-learning, the agent updates a table Q with entries $Q(s, a)$. When the agent transitions from state s at time t to new state s' at time $(t + 1)$, having taken action

a at time t, the table is updated according to:

$$\begin{aligned} Q_{k+1}(s,a) = Q_k(s,a) & \\ & + \alpha[r_t + \gamma \max_a Q_k(s',a) - Q_k(s,a)] \end{aligned} \tag{3.2}$$

where $0 \leq \alpha < 1$ is a learning rate and r_t is a reward [3]. The learning rate determines how important new information is to the estimate. A small value means that the new information will change the estimate only slightly, while a high value will alter it a great deal. This algorithm is known to converge if we assume that rewards are bounded $|r_t| \leq \mathcal{R}$, and the learning rate is restricted such that $0 \leq \alpha < 1$ and $\sum_{i=1}^{\infty} \alpha_{i(s,a)} = \infty, \sum_{i=1}^{\infty} [\alpha_{i(s,a)}]^2 < \infty, \forall s, a$ [28].

The intuitive meaning of a Q-value is different from that in Equation (3.1). The Q-table has an extra dimension to consider. Instead of referring to the estimated value of a state, we refer to the estimated value of performing an action a in a state s. For example, this means that instead of referring to the value of the game Roulette itself, we can refer to the value of performing different actions on the Roulette table, as seen in Figure 3.1. One possible action would be to bet on even numbers. In Roulette, the outcome is random so we expect to have nearly 50% chance of winning (not 50% because of the zero and double-zero). The payout (reward) for this action is 1 to 1, meaning that we would receive a payout equal to the wager, in case of success, and nothing, in case of failure. As the Q-table is updated, it learns the value of wagering on even and converges to the true value of doing so, which we expect to be worse than the value of staying home.

3.1.2 Introducing Variable Delays

In general, we cannot assume that the agent will receive rewards immediately after performing an action. Instead, let us consider that all rewards are delayed by a

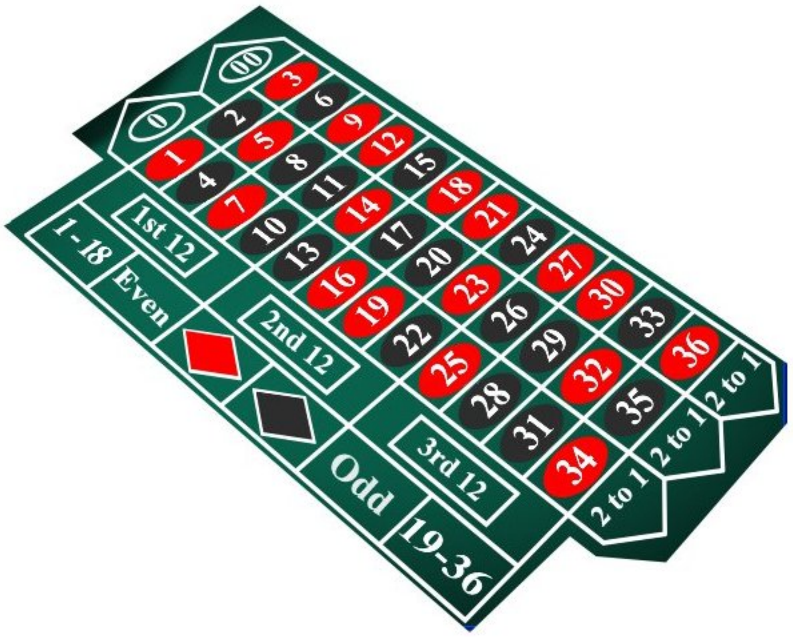


Figure 3.1: A Roulette table

random number of time steps d : Pois(λ) where:

$$\text{Pois}(d;\lambda) = \frac{\lambda^d}{d!}e^{-\lambda} \tag{3.3}$$

That is, d is Poisson distributed with expected value and variance λ. In this case, if we use Q-learning without modification, we will be assigning credit to the wrong state-action pairs with probability $1 - \text{Pois}(0;\lambda)$. Note that as λ grows, so does the likelihood of error. In Roulette, this would mean that we would not receive our payout until after d rounds had occurred. It would be difficult for us to know which wager had caused which payout if we did not know about the statistics of the delay beforehand. Figure 3.2 provides an example of the delay distribution when $\lambda = 5$. The vertical axis of this plot represents the outcome of a numerical experiment and is not a probability measure.

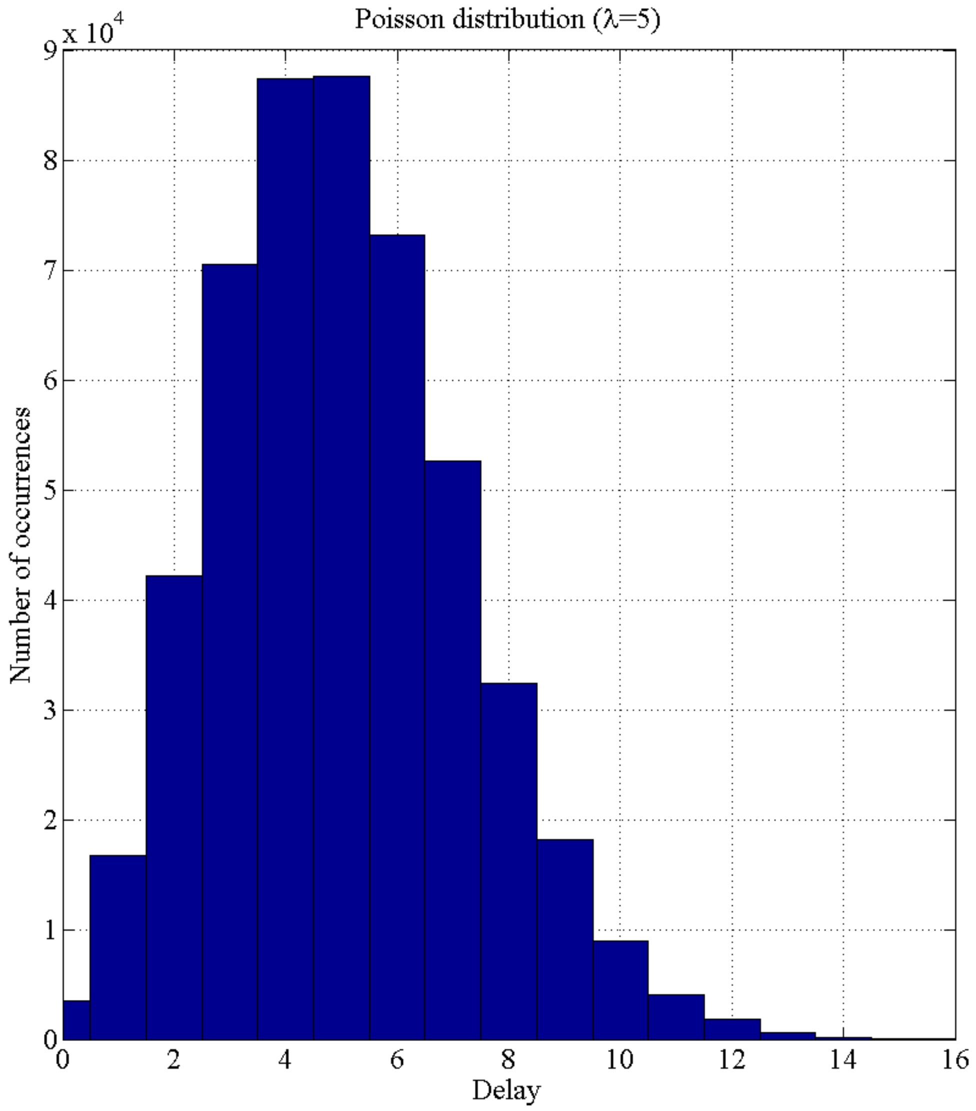


Figure 3.2: Delay distribution when $\lambda = 5$

3.2 Multiple Model Q-Learning

To address the situation where a variable delay is present in the reward signal, we propose an algorithm called Multiple Model Q-learning. Since it is possible to use RL to teach an agent about which states and actions are most valuable, it should also be possible to use RL to teach the agent about the statistics of the reward delay so that a valuable estimate of the delay can be determined.

Let $\hat{\lambda}$ be an estimate of the true mean and variance λ. We seek to modify Q-learning in a way that allows it to learn about the most valuable delay estimate $\hat{\lambda}$, which may tend towards the true value of λ.

To implement this idea, we will insert parallel Q-tables to be updated simultaneously. In Q-learning, the Q-table inherently assumes that the agent is receiving its rewards immediately, i.e. that $\lambda = 0$. When λ can take on other values, we can maintain parallel Q-tables which inherently assumes other possible values as well.

Theoretically, the Poisson distribution has no maximum value and so we must clip it in practice to maintain a reasonable number of parallel Q-tables. To that end, let $\hat{\lambda}_{max}$ be the designer-estimated maximum feasible value of λ. We do not expect the delay to be larger than this value. Even if it rarely exceeds this value, it will have negligible impact on the algorithm. We will now add the new learning dimension to the Q-table. Let us refer to entries in the new Q-table as:

$$Q(s, a, \hat{\lambda}) \tag{3.4}$$

where s is the state, a is the action, and $0 \leq \hat{\lambda} \leq \hat{\lambda}_{max}$ is the delay estimate. This new definition will allow the agent to do hypothesis testing as inspired by multiple-model control [48]. This means that the agent will consider several estimates of the delay simultaneously and automatically switch to the estimate which is most useful in every iteration.

Thus, the action-space is augmented with a set of delay estimates, Λ, which contains $|\Lambda| = \hat{\lambda}_{max} + 1$ estimates for the mean reward time delay, increasing the size of the action-space to $A \times |\Lambda|$. This assumes that we use an estimate resolution of one time-step, which is arbitrary. For each state, the agent must now select both an action and a delay estimate. The parallel Q-tables can be visualized as in Figure 3.3. In each time step, a candidate model is selected and its Q-table is made active for decision making during that time step. The dashed lines indicate inactive models which are still updated based on decisions made by the active model. In short, the active table drives behaviour, but inactive tables are still updated based on this behaviour.

Consider again the Roulette table of Figure 3.1. The gambler does not know about the statistics of the payout delay beforehand. Upon making a wager, the gambler must wait a mean of λ rounds to receive the outcome of that wager. If he nevertheless places wagers each round, he will have a delayed stream of outcomes queued to arrive in the future. They may arrive out of order or even overlap. Consider the example in Table 3.1. Wagers #1 and #3 have their outcomes overlap when received in round #3. Also, the result of wager #2 is not known until after the result of wager #3 is already received. We must develop a strategy to assign the payouts to the proper wagers, otherwise our estimates $Q(s, a)$ will be fraught with problems because they are based on the false assumption that the payout is immediate.

Table 3.1: Roulette delay example

Outcome	*Win*	Lose	*Win*	Lose	Lose	Lose	*Win*	*Win*
Round Wagered	1	2	3	4	5	6	7	8
Delay	2	5	0	6	10	4	5	5
Round Received	3	7	3	10	15	10	12	13

First, to begin to learn about the delay, the agent selects the most valuable time

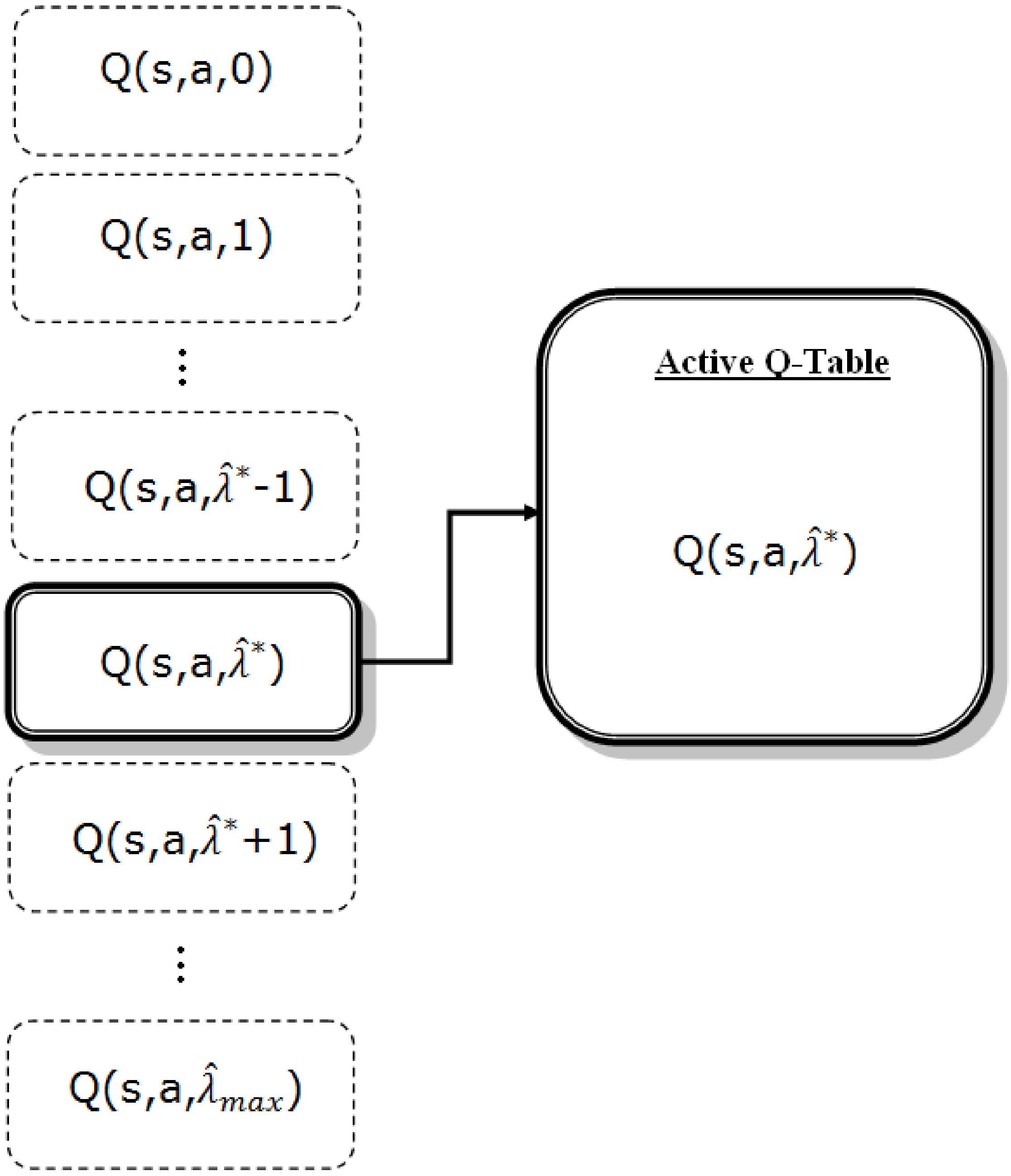

Q(s,a,0)
Q(s,a,1)
Q(s,a,$\hat{\lambda}^*$-1)
Q(s,a,$\hat{\lambda}^*$)
Q(s,a,$\hat{\lambda}^*$+1)
Q(s,a,$\hat{\lambda}_{max}$)
Active Q-Table
Q(s,a,$\hat{\lambda}^*$)

Figure 3.3: Activation of a candidate model in multiple model Q-Learning

delay estimate $\hat{\lambda}^*$ where:

$$\hat{\lambda}^* = \arg\max_{\hat{\lambda}} Q(s, a, \hat{\lambda}) \tag{3.5}$$

and then assumes that this estimate is correct. Then, as is normal in Q-learning, the agent selects the most valuable action according to:

$$a^* = \arg\max_{a} Q(s, a, \hat{\lambda}^*) \tag{3.6}$$

given that the agent is currently in state s under the assumption that $\lambda = \hat{\lambda}^*$. The agent then performs action a^* and arrives at state s'. A reward is produced, but the agent will not receive this reward until the stochastic delay has elapsed.

During this time step t, a reward is received which may be the sum of several rewards (caused by several actions), a single reward (caused by one action) or no rewards at all (a reward of zero). Overlap is possible as a result of the relaxation of the assumption that the rewards are delivered in proper order. For example, a reward may be delayed by six time steps and then the next reward is delayed by five time steps. In five time steps, the agent would receive the sum of the two rewards.

Since we do not know the true value of the mean delay, we assign credit according to all estimates simultaneously. To do so, the agent cycles through all $|\Lambda|$ estimate models and assigns the reward at time step t to the state-action pair at time step $t - \hat{\lambda}$ such that:

$$\begin{aligned} Q_{k+1}(s_{t-\hat{\lambda}}, a_{t-\hat{\lambda}}, \hat{\lambda}) = {} & Q_k(s_{t-\hat{\lambda}}, a_{t-\hat{\lambda}}, \hat{\lambda}) \\ & + \alpha \left[r_t + \gamma \max_{\hat{\lambda}, a} Q_k(s'_{t-\hat{\lambda}}, a, \hat{\lambda}) \right] \\ & - \alpha \left[Q_k(s_{t-\hat{\lambda}}, a_{t-\hat{\lambda}}, \hat{\lambda}) \right] \end{aligned} \tag{3.7}$$

where $\hat{\lambda}$ is the $\hat{\lambda}^{\text{th}}$ model of $|\Lambda|$ total models. To summarize, the agent acts according to the best delay estimate, but updates all delay estimates based on the outcome. Each update is independent and so these computations can be done in parallel to speed up the algorithm. In this way, the agent repeatedly evaluates $|\Lambda|$ different estimates at once even though it cannot act according to them all simultaneously. The agent then chooses the next best candidate estimate model $\hat{\lambda}^*$ and the cycle continues until sufficient exploration of the environment is achieved. For example, when the agent is updating $Q(s, a, 2)$, it is inherently assuming that the delay is equal to two time steps. All credit is thus assigned to the state-action pair two time steps in the past because if the delay is actually two time steps, then this state-action pair must be the cause.

Over time, some estimates will prove to be much more valuable than others. This is because for each value of $\hat{\lambda}$, there is a Q-table $Q(s, a)$. Each of these Q-tables assumes that the reward delay is constant and equal to $\hat{\lambda}$. In an undelayed case, it is assumed that rewards arrive immediately and so credit can be assigned with certainty to the proper cause. With a variable delay, it is no longer possible to assign credit properly 100% of the time. Instead, we must settle for some mistakes and strive to minimize the chance of them occurring because assigning credit under the assumption that $\lambda = \hat{\lambda}$ will only be correct a fraction of the time equal to:

$$\text{Pois}(\hat{\lambda}; \lambda) = \frac{\lambda^{\hat{\lambda}}}{\hat{\lambda}!} e^{-\lambda} \tag{3.8}$$

We therefore attempt to assume a constant delay which minimizes the chance of incorrect assignment. The Q-tables which assume poor delay estimates will produce behaviour that is undesirable because the policy will be misinformed. Over time, this low quality behaviour will cause the poor delay estimate to have a lower estimated value. Eventually, when the value falls too low, a new estimate will be used to determine behaviour instead. This process continues until the delay estimates have been sufficiently explored and the most valuable estimate has been found.

For example, if the mean delay is $\lambda = 5$, but we repeatedly place Roulette wagers based on the assumption that the delay is a constant twelve time steps, then our value estimates for different Roulette actions will be poorly informed and inaccurate. Instead, if we select a delay estimate closer to five, we will be assigning the payouts to the correct actions correctly more frequently, although we will still be incorrect some fraction of the time. Thus, given the same amount of information, our policy will never be as useful as the policy which could be created in the undelayed case, where no assignment errors are made. However, we should expect to create a better policy than Q-learning would if a delay is present because Q-learning does not have the capability of adjusting its fundamental assumption that rewards arrive immediately.

We now understand how the delay estimates are treated, but how do we actually select actions? When the agent acts, it must select an action from one of the parallel Q-tables, as in Figure 3.3. To select an action, the agent must first choose an active delay estimate $\hat{\lambda}$, as described above. Then, the action is chosen from this Q-table as is normally done in Q-learning. Once the agent has an active table, it behaves as though this is the only table, although it still updates the inactive tables with the results of its behaviour. Over time, different tables become active as the agent explores the delay estimate space.

Algorithm 1 contains the full algorithm in pseudo-code. As discussed in Chapter 2, optimistic initialization and ϵ-greedy are techniques often used in Q-learning to balance the exploration and exploitation of the environment.

Let us trace the algorithm using a simple Roulette example. Assume that a gambler's only options are to bet even or to bet on #13. We are only playing Roulette, so the state is always the same. We can create a 1×2 Q-table to represent these options. Assume also that we always wager \$1. We begin by setting $\hat{\lambda}_{\max} = 2$ because

Algorithm 1 - Multiple Model Q-Learning

```
Set $\hat{\lambda}_{\max}$
Initialize state-action memory of sufficient length
Initialize $\alpha, \gamma$ (e.g. $\alpha = 0.1, \gamma = 0.9$)
Initialize $Q(s, a, \hat{\lambda})$ arbitrarily (e.g. optimistic initialization)
for episode do
    Initialize $s$
    repeat
        select $\hat{\lambda}^*$ using policy from Q (e.g. $\epsilon$-greedy)
        select $a^*$ using policy from Q assuming $\hat{\lambda}^*$ is correct (e.g. $\epsilon$-greedy)
        execute action $a^*$
        observe $r_t$ and $s'$
        for $\hat{\lambda}$ from 0 to $\hat{\lambda}_{max}$ do
            $$Q_{k+1}(s_{t-\hat{\lambda}}, a_{t-\hat{\lambda}}, \hat{\lambda}) \leftarrow (1-\alpha)Q_k(s_{t-\hat{\lambda}}, a_{t-\hat{\lambda}}, \hat{\lambda}) + \alpha \left[ r_t + \gamma \max_{\hat{\lambda}, a} Q_k(s'_{t-\hat{\lambda}}, a, \hat{\lambda}) \right]$$
        end for
        $s \leftarrow s'$
    until $s$ is terminal or behaviour is acceptable
end for
```

we are confident that the delay will never be larger than two. This allows for three possible delay estimates, so our table has dimension $1 \times 2 \times 3$. To assign credit two steps into the past, we must have a memory of the current state and action, as is normal in Q-learning, plus a memory of the previous two states and actions. We then set typical values $\alpha = 0.1$ and $\gamma = 0.9$. Next, we choose to initialize the Q-table optimistically. We set all values to 50 because we think that this value is too high and will encourage exploration.

Table 3.2: Initial Roulette Q-table

	$\hat{\lambda} = 0$		$\hat{\lambda} = 1$		$\hat{\lambda} = 2$	
	Even	**13**	**Even**	**13**	**Even**	**13**
Roulette	50.00	50.00	50.00	50.00	50.00	50.00

Now we begin the first Roulette round. We randomly choose $\hat{\lambda}^* = 1$ because all Q-values are set to 50, as in Table 3.2. Now we go to Q-table $Q(s, a, 1)$ and select an action ϵ-greedily. It is a tie, so we choose action one randomly. We place a \$1 wager on even. The payout happens to be delayed by two time steps. Our immediate payout is zero, because we do not know the result of our first wager yet. We update the Q-table to be as Table 3.3. Only $Q(s, a, 0)$ can be updated because previous actions do not exist.

Table 3.3: Roulette Q-table after round 1

	$\hat{\lambda} = 0$		$\hat{\lambda} = 1$		$\hat{\lambda} = 2$	
	Even	**13**	**Even**	**13**	**Even**	**13**
Roulette	49.50	50.00	50.00	50.00	50.00	50.00

The second Roulette round begins. Say that we end up with $\hat{\lambda}^* = 1, a^* = 2$. We place a bet on #13. The payout of this wager happens to be delayed by zero time steps, and so we receive it immediately. We won and received a payout of \$35. We update the Q-table to be as Table 3.4.

Table 3.4: Roulette Q-table after round 2

	$\hat{\lambda} = 0$		$\hat{\lambda} = 1$		$\hat{\lambda} = 2$	
	Even	**13**	**Even**	**13**	**Even**	**13**
Roulette	49.50	53.00	53.00	50.00	50.00	50.00

For $\hat{\lambda} = 0$, we assign the payout to the wager on #13. For $\hat{\lambda} = 1$, we assign the payout to the wager on even. We cannot update $\hat{\lambda} = 2$ yet because of insufficient history. Notice that the update for $\hat{\lambda} = 1$ did not use the new max value (53) generated by the update of $\hat{\lambda} = 0$. This is so that the calculations can be done in parallel. If not done in parallel, the most up-to-date values can be used without problem.

Round three begins. So far, the Q-table estimates that the best delay estimate is either $\lambda \hat{=} 0$ or $\hat{\lambda} = 1$ because they contain actions with values of 53. Let us say that the policy chooses $\hat{\lambda}^* = 1, a = 1$. We place a \$1 wager on even because this is the best action according to the Q-table with delay estimate $\hat{\lambda}^*$. The outcome is delayed by one time step. However, we receive the outcome from our round 1 wager. We lost, and receive $-\$1$. The Q-table is updated to be as Table 3.5.

Table 3.5: Roulette Q-table after round 3

	$\hat{\lambda} = 0$		$\hat{\lambda} = 1$		$\hat{\lambda} = 2$	
	Even	**13**	**Even**	**13**	**Even**	**13**
Roulette	49.22	53.00	53.00	49.67	49.67	50.00

First, $\hat{\lambda} = 0$ is updated, which assumes that the round 3 wager on even caused the present payout of $-\$1$. Next, $\hat{\lambda} = 1$ is updated, which assumes that the round 2 wager on #13 caused the current payout. Finally, $\hat{\lambda} = 2$ is updated, which assumes that the round 1 wager on even caused the current payout. The update process continues like this until the Q-table converges to reflect a best estimate of the value of the actions. Of course, this would take many iterations.

Chapter 4

Theoretical Results

This section contains the theoretical results of this thesis which include a proof of convergence for the multiple-model Q-learning algorithm.

4.1 Convergence

We will use a convergence theorem found in [16] to prove convergence for multiple model Q-learning in similar fashion as has been done for Q-learning. In that work, the author provided a more general proof of convergence for Q-learning, as was originally shown to converge in [28]. Q-learning has also been shown to converge in [49]. The present proof will show convergence in the stochastic sense because Q-learning is inherently a stochastic approximation based on the expected value of state-action pairs within a particular environment. To parallel the proof in [16], we will refer to the minimization of costs instead of the maximization of rewards during the proof. This is simply the dual problem.

To begin, we reiterate the model for the RL problem which is to be solved by the multiple model Q-learning algorithm. Consider a Markov decision problem defined on finite state space S. For every state $s \in S$, there is a finite set $A(s)$ of possible control actions and a set of nonnegative scalars $p_{ss'}(a), a \in A(S), s' \in S$, such that

$\sum_{s' \in S} p_{ss'}(a) = 1$ for all $a \in A(s)$. The scalar $p_{ss'}(a)$ is interpreted as the probability of a transition to s', given that the current state is s and control a is applied. Also, for every state s and control a, there is a random variable $r(t)$ which represents the one-stage cost at time step t, which may be caused by the overlap of any number of past state-action pairs. This cost is then delayed by an amount d which is Poisson distributed with mean and variance λ. We assume that the variance of $r(t)$ is finite for every s and $a \in A(s)$. Finally, we maintain a current delay estimate $\hat{\lambda}$ from the set of delay estimates Λ.

In addition, let t be a discrete time variable which acts as an index for updates. $x(t)$ is the value of the vector x at time t and $x_i(t)$ is the i^{th} component of x. We represent the set of time indices at which x_i is updated as T^i. This implies that:

$$x_i(t+1) = x_i(t) \qquad t \notin T^i \tag{4.1}$$

which means that the element i of vector x is unchanged at any time index not contained in T^i. For example, entry x_1 may have been updated only at times $t = 1, 3, 4, 7, ...$ during learning. It is therefore possible that some components of x have been updated more recently than others. In other words, at time t, we may not have relevant experience to update all entries at once. Only a single state-action pair is used at any instant.

Next, we define a form for the update equation which facilitates learning:

$$x_i = x_i + \alpha_i(t)(F_i(x^i(t)) - x_i + w_i) \tag{4.2}$$

Note that $\alpha_i(t)$ is a stepsize parameter in $[0, 1]$, $w_i(t)$ is a noise term, and $F(x)$ is a mapping from $\mathbb{R}^n$ to itself. The value of $\alpha_i(t)$ determines how important each successive update is. A low value means that the value of $x_i(t)$ will change little, while a high value will cause the present update to have more impact. Thus, we can control the relative impact of new experience. $x^i(t)$ represents a vector of components of x, which may or may not be outdated. It can be expressed as:

$$x^i(t) = [x_1(\tau_1^i(t)), ..., x_n(\tau_n^i(t))], \qquad t \in T^i, \tag{4.3}$$

where each $\tau_j^i(t)$ is an index which is either a current or historical update index such that $0 \leq \tau_j^i(t) \leq t$. For example, if $t = 6$, then we might have $\tau_1^i(6) = 4$, which means that we are using the old value of x_1 from the update at $t = 4$, even if there was a newer update at $t = 5$. This is all to illustrate that it is permissible to use outdated information in the updates. A special case is when all $\tau_j^i(t) = t \ \forall \ i, j$, and t. In this scenario, $x^i(t) = x(t)$, which means that the current values of $x(t)$ are being used to calculate $F_i(x^i(t))$. When this is true, the update equation only relies on the current value of $x(t)$ and need not remember any further past values to calculate $x(t+1)$.

We aim to prove that the Q-table converges to a fixed point. That is, the entries $Q_{s\hat{\lambda}a} \ \forall \ s, \hat{\lambda}, s$ all converge to a final Q-table, Q^*. In this proof, we are using the general vector x to refer to the information stored in what might be a Q-table. We now introduce the theorem to be used for convergence, which states that there are four necessary assumptions for an update equation of the proposed form to cause convergence. We will then present these assumptions, before demonstrating that they are satisfied by multiple model Q-learning.

Theorem *If the following four assumptions are true, and if the algorithm updates the vector x using the stochastic approximation structure $x_i = x_i + \alpha(F_i(x^i) - x_i + w_i)$, then $x(t)$ converges to x^* with probability 1* [16].

The first assumption states that as our time index t approaches infinity, all time indices used to generate $x^i(t)$ will also approach infinity. Although information can be outdated, this means that any old information will eventually be replaced in the update equation. In other words, there is some point where all old information will cease to be used.

Assumption 1 *For any i and j, $\lim_{t\to\infty} \tau_j^i(t) = \infty$, with probability 1.*

The second assumption uses the notation $\mathcal{F}(t)$, referring to the history of the algorithm up to just before $x(t+1)$ is computed. If something is said to be $\mathcal{F}(t)$-measurable, it means that it can be entirely determined by the history referred to by $\mathcal{F}(t)$. The assumption defines the amount of history required to determine the different parameters in the update equation and defines the statistics of the noise term $w_i(t)$. The assumption implies that it is possible to decide whether to update some component x_i at time t based on the history up to that point. Later, we will show how, point by point, multiple model Q-learning satisfies this assumption.

Assumption 2 *Any random variables are defined within a probability space $(\Omega, \mathcal{F}, P)$ and there exists an increasing sequence $[\mathcal{F}(t)]_{t=0}^{\infty}$ of subfields of $\mathcal{F}$ such that:*

a) x(0) is $\mathcal{F}(0)$-measurable;

b) For every i and t, $w_i(t)$ is $\mathcal{F}(t+1)$-measurable;

c) For every $i, j,$ and $t, \alpha_i(t)$ and $\tau_j^i(t)$ are
$\mathcal{F}(t)$-measurable.

d) For every i and t, we have $E[w_i(t)|\mathcal{F}(t)] = 0$;

e) and there exist (deterministic) constants A and B such that

$$E[w_i^2(t)] \leq A + B \max_j \max_{\tau \leq t} |x_j(\tau)|^2 \, \forall i, t$$

The third assumption is a standard stepsize condition for stochastic convergence. It places restrictions on the choice of learning parameter. Typically, this parameter is set so that it decreases and tends to zero over time. This embodies the idea that experience should be more important when the learning agent has less accumulated knowledge and less important when the agent has become accustomed to the environment. Intuitively, the value of α should never go to zero (learning should never cease) but α must not be too large and should generally decrease over time, although it is not necessary to decrease monotonically.

Assumption 3 *For every s,*

$$\sum_{t=0}^{\infty} \alpha_s(t) = \infty, \ \textit{w.p.1.} \tag{4.4}$$

There exists some (deterministic) constant C such that for every s,

$$\sum_{t=0}^{\infty} \alpha_s^2(t) \leq C, \ \textit{w.p.1.} \tag{4.5}$$

The final assumption places conditions on F, the choice of mapping. Later, we will define F in terms of multiple model Q-learning. Basically, we are forcing F to be a contraction mapping.

Assumption 4 *Defining the norm $||\cdot||_v$ on $\mathbb{R}^n$ as*

$$||x||_v = \max_i \frac{|x_i|}{|v_i|} \tag{4.6}$$

there exists a vector $x^* \in \mathbb{R}^n$, a positive vector v, and a scalar $\gamma \in [0, 1)$, such that

$$||F(x) - x^*||_v \leq \gamma ||x - x^*||_v \qquad \forall x \in \mathbb{R}^n \tag{4.7}$$

We begin by defining the dynamic programming operator $T : \mathbb{R}^{|S|} \mapsto \mathbb{R}^{|S|}$, with components T_s. Let

$$T_s(V) = \min_{\hat{\lambda}\in\Lambda, a\in A(s)} \left(E[r_{(sa)_d}] + \gamma \sum_{s'\in S} p_{ss'}(a) V_{s'} \right) \tag{4.8}$$

where V_s is an estimate of the value of state s.

In general, it is known that if $\gamma < 1$, then T is a contraction with respect to the norm $||\cdot||_\infty$, with V^* being its unique fixed point. $E(r_{(sa)_d})$ is the expected value of cost generated by the state action pair (s, a) d time steps ago.

Remark 1 *Multiple model Q-learning is a modification of Q-learning, which is itself a modification of the Bellman equation $V^* = T(V^*)$ [16].*

When costs are used instead of rewards, multiple model Q-learning updates according to:

$$\begin{aligned} Q_{s\hat{\lambda}a}(t+1) = Q_{s\hat{\lambda}a}(t) &+ \alpha_{s\hat{\lambda}a}(t)\left[r_{(sa)_d}\right] \\ &+ \alpha_{s\hat{\lambda}a}(t)\left[\gamma \min_{\hat{\lambda}'\in\Lambda, a'\in A(S(s,a))} Q_{(S(s,a),\hat{\lambda}',a')}(t)\right] \\ &- \alpha_{s\hat{\lambda}a}(t)\left[Q_{s\hat{\lambda}a}(t)\right] \end{aligned} \tag{4.9}$$

where $r_{(sa)_d}$ is the cost generated by the state-action pair which occurred at time step $(t-d)$. However, we are assuming that this cost was generated at time step $(t-\hat{\lambda})$. When we know that $\lambda = 0$, then $d = 0$, as is assumed in Q-learning, this update equation becomes the special case:

$$\begin{aligned} Q_{sa}(t+1) = Q_{sa}(t) &+ \alpha_{sa}(t)\left[r(t)\right] \\ &+ \alpha_{sa}(t)\left[\gamma \min_{a'\in A(S(s,a))} Q_{S(s,a),a'}(t) - Q_{sa}(t)\right] \end{aligned} \tag{4.10}$$

where $S(s, a)$ is a random successor state which is equal to s' with probability $p_{ss'}(a)$.

This is the case where we can assign the costs with certainty because we know the delay with certainty. When $\lambda \neq 0$, this will not be the case. Equation (4.10) has already been shown to converge in [16]. We must extend the proof to prove the convergence of Equation (4.9).

Theorem *$Q_{s\hat{\lambda}a}(t)$ converges to $Q^*_{s\hat{\lambda}a}$ with probability 1.*

Proof *We must now show that multiple model Q-learning, defined by Equation* (4.9)*, takes on the form:*

$$x_i(t+1) = x_i(t) + \alpha_i(t)\left(F_i(x_i(t)) - x_i(t) + w_i(t)\right)$$
$$t \in T^i \tag{4.11}$$

where F is a mapping from $\mathbb{R}^n$ into itself with components $F_i \equiv F_{s\hat{\lambda}a}$ and that it satisfies the four assumptions. The subscript $s\hat{\lambda}a$ is simply a more compact notation for entry $(s, \hat{\lambda}, a)$ in the table. We first define:

$$F_{s\hat{\lambda}a}(Q) = E[r_{(sa)_d}(t)] + \gamma E\left[\underset{\hat{\lambda}' \in \Lambda, a' \in A(S(s,a))}{min} Q_{S(s,a),\hat{\lambda}',a'}\right] \tag{4.12}$$

and because $p_{ss'}(a)$ is the probability of moving from state s to successor state s' after having taken action a, we can write the expected minimum Q value by summing over all possible successor states:

$$E\left[\underset{\hat{\lambda}' \in \Lambda, a' \in A(S(s,a))}{min} Q_{S(s,a),\hat{\lambda}',a'}\right] = \sum_{s' \in S} p_{ss'}(a) \underset{\hat{\lambda}' \in \Lambda, a' \in A(s')}{min} Q_{s'\hat{\lambda}'a'}$$

Now we have defined the mapping F. We have designed F in this way because it must be a contraction mapping with fixed point Q^*. Note that if a vector Q^* is *a fixed point of the mapping F, i.e. $F(Q^*) = Q^*$, then the vector with components $V_s = \min_{\hat{\lambda}\in\Lambda, a\in A(s)} Q_{s\hat{\lambda}a}$ is a fixed point of T because, from Equation* (4.8)*:*

$$\begin{aligned}
T_s(V) &= \min_{\hat{\lambda}\in\Lambda, a\in A(s)} \left(E[r_{(sa)_d}] + \gamma \sum_{s'\in S} p_{ss'}(a) V_{s'} \right) \\
&= \min_{\hat{\lambda}\in\Lambda, a\in A(s)} E[r_{(sa)_d}] \\
&+ \min_{\hat{\lambda}\in\Lambda, a\in A(s)} \left(\gamma \sum_{s'\in S} p_{ss'}(a) \min_{\hat{\lambda}\in\Lambda, a\in A(s)} Q_{s\hat{\lambda}a} \right) \\
&= \min_{\hat{\lambda}\in\Lambda, a\in A(s)} F(Q_{s\hat{\lambda}a}) \\
&= \min_{\hat{\lambda}\in\Lambda, a\in A(s)} Q_{s\hat{\lambda}a} \\
&= V
\end{aligned}$$

After substituting the definition of F from Equation (4.12) *into Equation* (4.9)*, we can now rewrite the update equation,* (4.9) *as:*

$$\begin{aligned}
Q_{s\hat{\lambda}a}(t+1) = Q_{s\hat{\lambda}a}(t) \\
+ \alpha_{s\hat{\lambda}a}(t) \left[F_{s\hat{\lambda}a} \left(Q_{s\hat{\lambda}a}(t) \right) \right] \\
+ \alpha_{s\hat{\lambda}a}(t) \left[-Q_{s\hat{\lambda}a}(t) + w_{s\hat{\lambda}a}(t) \right]
\end{aligned} \tag{4.13}$$

We defined F to suit our needs. Next, we define the noise term w so that $F + w =$

$r_{(sa)_d} + \gamma \min_{\hat{\lambda}' \in \Lambda, a' \in A(S(s,a))} Q_{(S(s,a),\hat{\lambda}',a')}(t)$, *as in Equation* (4.9).

$$\begin{aligned} w_{s\hat{\lambda}a}(t) = & \; r_{(sa)_d}(t) - E[r_{(sa)_d}(t)] \\ & + \gamma \min_{\hat{\lambda}' \in \Lambda, a' \in A(S(s,a))} Q_{(S(s,a),\hat{\lambda}',a')}(t) \\ & - \gamma E\left[\min_{\hat{\lambda}' \in \Lambda, a' \in A(S(s,a))} Q_{(S(s,a),\hat{\lambda}',a')}(t)|\mathcal{F}(t)\right] \end{aligned} \tag{4.14}$$

The expectation is with respect to $S(s,a)$. *When we substitute the definitions of* F *and* w *into Equation* (4.13), *we arrive back at Equation* (4.9). *However, we can now express the update equation in the desired form from Equation* (4.11) *because we have defined* F *and* w.

Using these definitions, the next step is to show that the convergence theorem's four assumptions are satisfied.

Assumption 1 *is satisfied because there is no maximum time index. Because we have theoretically infinite time to perform updates, we can eventually discard all outdated information. As* $t \to \infty$, *all* $\tau_j^i(t) = \infty$ *for any* i *and* j. *Intuitively, this means that we eventually only use time indices which are not less than the time index of the update itself. In other words, no information being used in the update equation is out of date. The choice of data to be used for performing the update is up to the designer, and so we can be sure to satisfy this assumption. The notion of* $\tau_j^i(t) \neq t$ *only allows us to use outdated information in the updates, but it does not require that we do so. We may use outdated information to perform the updates for an arbitrary length of time, as long as we eventually cease to do so for all elements in the Q-table.*

Assumption 2 *contains several parts. Part (a) is automatically valid because if* $x(0)$ *is initialized, it is* $\mathcal{F}(0)$*-measurable. Part (b) is also valid because* $w_{s\hat{\lambda}a}(t)$ *is defined only using values which are* $\mathcal{F}(t)$*-measurable, as in Equation* (4.14). *Part (c) can be satisfied because the values* $\alpha_i(t)$ *and* $\tau_j^i(t)$ *are* $\mathcal{F}$*-measurable, since they* *are entirely specified by us and so we allow the needed values to be generated before updating the components which require them. For example, we can choose the learning rate* α *on the fly, but we do not need to know information from the future to do so. To satisfy part (d) we can show that the expected value of the noise term is zero. From Equation* (4.14)*, and for every state* s *and time index* t*:*

$$\begin{aligned} E[w_{s\hat{\lambda}a}(t)|\mathcal{F}(t)] &= E[r_{(sa)_d}(t)] - E[r_{(sa)_d}(t)] \\ &+ \gamma E\left[\min_{\hat{\lambda}'\in\Lambda, a'\in A(S(s,a))} Q_{(S(s,a),\hat{\lambda}',a')}(t)|\mathcal{F}(t)\right] \\ &- \gamma E\left[\min_{\hat{\lambda}'\in\Lambda, a'\in A(S(s,a))} Q_{(S(s,a),\hat{\lambda}',a')}(t)|\mathcal{F}(t)\right] \\ &= 0 \end{aligned}$$

Finally, part (e) requires that the conditional variance $E[w_i^2(t)]$ *be bounded by* $A + B \max_{s'\in S} \max_{\hat{\lambda}'\in\Lambda} \max_{a'\in A(S(s,a))} \max_{\tau\le t} |Q_{s'\hat{\lambda}'a'}(\tau)|^2 \,\forall s, \hat{\lambda}, a, t$*. From Equation* (4.14)*, and taking one term at a time, we will now compute a bound for* $E[w^2_{s\hat{\lambda}a(t)}|\mathcal{F}(t)]$*. We already know that* $E[w_{s\hat{\lambda}a}] = 0$*, so we can compute the variance as* $\mathrm{Var}(w_{s\hat{\lambda}a}|\mathcal{F}(t)) = E[(w_{s\hat{\lambda}a} - 0)^2] = E[w^2_{s\hat{\lambda}a}]$*.*

If we let

$$a = r_{(sa)_d}(t) - E[r_{(sa)_d}(t)] \tag{4.15}$$

$$\begin{aligned} b = \gamma &\min_{\hat{\lambda}'\in\Lambda, a'\in A(S(s,a))} Q_{(S(s,a),\hat{\lambda}',a')}(t) \\ &- \gamma E\left[\min_{\hat{\lambda}'\in\Lambda, a'\in A(S(s,a))} Q_{(S(s,a),\hat{\lambda}',a')}(t)|\mathcal{F}(t)\right] \end{aligned} \tag{4.16}$$

and, referring to Equation (4.14)*, we calculate* $E[w^2_{s\hat{\lambda}a}]$ *to be* $E[a^2+ab+b^2]$*. The first term,* $E[a^2]$ *is the variance of the cost function,* $\mathrm{Var}[r_{(sa)_d}(t)]$*. The third term is the* *variance of the minimization term,* $\gamma \underset{\hat{\lambda}' \in \Lambda, a' \in A(S(s,a))}{min} Q_{(S(s,a),\hat{\lambda}',a')}(t)$*. We know that the maximum value of the variance of the minimization term must be no higher than*

$$B = \max_{s' \in S} \max_{\hat{\lambda}' \in \Lambda} \max_{a' \in A(S(s,a))} \max_{\tau \leq t} |Q_{s'\hat{\lambda}'a'}|^2(\tau)$$

because this is the square of the largest Q-value that has ever been in the table up to time t*.*

The second term is the expected value of the product (ab)*. We expand this term. For brevity, let* $r = r_{(sa)_d}(t)$ *and* $Q = \gamma \underset{\hat{\lambda}' \in \Lambda, a' \in A(S(s,a))}{min} Q_{(S(s,a),\hat{\lambda}',a')}(t)$*.*

$$\begin{aligned} E[ab] &= E[(r - E[r])(Q - E[Q])] \\ &= E[rQ - rE[Q] - E[r]Q + E[r]E[Q]] \\ &= E[rQ] - E[rE[Q]] + E[r]E[Q] - E[E[r]Q] \end{aligned}$$

but the second and third term are equivalent and cancel. We are left with

$$\begin{aligned} E[ab] &= E[rQ] - E[E[r]Q] \\ &= E[rQ] - E[r]E[Q] \end{aligned}$$

If r *and* Q *are independent, then this also reduces to zero because* $E[rQ] = E[r]E[Q]$*. If they are not independent, then it still provides a deterministic upper bound* A*. Thus, we can be sure to bound the variance of the noise term by summing the following four* *terms.*

$$\begin{aligned} E[w^2_{s\hat{\lambda}a}(t)|\mathcal{F}(t)] \leq\ & \mathrm{Var}[r_{(sa)_d}(t)] \\ & + E\left[r_{(sa)_d}(t)\gamma \min_{\hat{\lambda}'\in\Lambda, a'\in A(S(s,a))} Q_{(S(s,a),\hat{\lambda}',a')}(t)\right] \\ & - E\left[r_{(sa)_d}(t)\right] E\left[\gamma \min_{\hat{\lambda}'\in\Lambda, a'\in A(S(s,a))} Q_{(S(s,a),\hat{\lambda}',a')}(t)\right] \\ & + \max_{s'\in S} \max_{\hat{\lambda}'\in\Lambda} \max_{a'\in A(S(s,a))} \max_{\tau\leq t} |Q_{s'\hat{\lambda}'a'}|^2(\tau) \end{aligned}$$

Therefore, Assumption 2 is fulfilled.

Assumption 3 *is fulfilled because we may specify appropriate values for* $\alpha_s(t)$*. This is a design parameter and is completely in our control.*

Assumption 4 *is straightforward to fulfil when* $\gamma < 1$*. From Equation* (4.12)*:*

$$\begin{aligned} ||F_{s\hat{\lambda}a}(Q) - F_{s\hat{\lambda}a}(Q')||_\infty \leq\ & \gamma \min_{s\in S, \hat{\lambda}\in\Lambda, a\in A(s')} ||Q_{s\hat{\lambda}a} - Q'_{s\hat{\lambda}a}||_\infty \\ & \forall Q, Q' \end{aligned} \tag{4.17}$$

Substituting Equation (4.12)*, and examining the left-hand side of Equation* (4.17)*:*

$$\begin{aligned} \mathrm{LHS} =\ & \|E\left[r_{(sa)_d}(t)\right] + \gamma E\left[\min_{\hat{\lambda}'\in\Lambda, a'\in A(S(s,a))} Q_{(S(s,a),\hat{\lambda}',a')}(t)\right] \\ & - E[r_{(sa)_d}(t)] - \gamma E\left[\min_{\hat{\lambda}'\in\Lambda, a'\in A(S(s,a))} Q'_{(S(s,a),\hat{\lambda}',a')}(t)\right] \|_\infty \\ & = \gamma \left\| E\left[\min_{\hat{\lambda}'\in\Lambda, a'\in A(S(s,a))} Q_{(S(s,a),\hat{\lambda}',a')}(t)\right] - E\left[\min_{\hat{\lambda}'\in\Lambda, a'\in A(S(s,a))} Q'_{(S(s,a),\hat{\lambda}',a')}(t)\right]\right\|_\infty \end{aligned}$$

On the left-hand side, we have a maximum of a minimum, and on the right-hand side, we have a minimum of a maximum. Therefore, the left-hand side is always less than *or equal to the right-hand side value of*

$$\text{RHS} = \gamma \min_{s \in S, \hat{\lambda} \in \Lambda, a \in A(s')} \|Q_{s\hat{\lambda}a} - Q'_{s\hat{\lambda}a}\|_\infty \tag{4.18}$$

The mapping F was initially designed to be a contraction mapping, with respect to the maximum norm $\| \cdot \|_\infty$, and so this satisfies assumption 4.

The four assumptions in Theorem 1 are satisfied. Therefore, the Q-table $Q_{s\hat{\lambda}a}(t)$ converges to the fixed point $Q^*_{s\hat{\lambda}a}$ with probability 1.

□

Chapter 5

Simulation and Experimental Results

In this chapter, we will present both simulated and experimental results. These tests will show that the problem of stochastic delayed reinforcements can inhibit learning, even in simple environments. As well, we will show that multiple model Q-learning is able to significantly improve performance by reducing the effect of the reinforcement delay. We demonstrate its efficacy in grid world simulations, Webots simulations, and real world environments.

5.1 Grid world simulations

In this section, we conduct simulations in which we place an agent in a simple grid world environment to examine the effects of introducing variable reward delay into its learning algorithm. Figure 5.1 contains an example of a 3×3 grid world. Note that the start position is arbitrary. In this environment, the agent receives rewards of zero for all moves unless it comes into contact with a wall, where it receives a reward of -1. If the agent reaches the reward state in the top left, it receives a reward of $+10$ and is warped to the bottom right-most state.

We now provide an example of a possible path that the agent might follow during a run. Say that the agent begins in the bottom-right corner (this is arbitrary). It then has four possible actions (up, right, down, left). If the agent moves down or right, it bumps into the wall and generates a reward of -1. When a wall is bumped, the agent remains in the same state to avoid leaving the grid world. This reward is then delayed by some amount d which is Poisson distributed such that:

$$\text{Pois}(d; \lambda) = \frac{\lambda^d}{d!} e^{-\lambda} \tag{5.1}$$

where λ is the mean and variance of the distribution. If it hit the wall, the agent is now still in the bottom-right corner. Otherwise, it has moved up or left to the next square (state). The next state, in turn, generates its reward (which is zero for a non-wall) and the agent progresses until it reaches the top-left corner. When the agent chooses any action in the top-left corner, it is transported to the bottom-right corner to begin again and a reward of $+10$ is generated, and then delayed in time by d time steps. The path that optimizes the reward is any path composed of up actions and left actions which deliver the agent to the reward state in the top-left. In the 3 x 3 grid world, the minimum amount of steps to reach the reward state and return to the bottom-left is five. For example, the agent might choose (left, left, up, up, any) or (left, up, up, left, any).

The larger grid worlds used in this section work in the same way except that they contain more empty space within their walls, meaning that it is easier for the agent to become lost. They are of sizes 5×5 and 9×9. In these grid worlds, we will explore the effects of varying mean delays, varying world sizes, and experiment with credit assignment techniques for the purpose of building up to proposed solution to the delayed reinforcement problem.

To use Q-learning in these grid worlds, the agent must have a Q-table to update. The Q-tables are easy to discretize in a grid world scenario. For example, in a 3 x 3 grid world, there are nine possible states and four possible actions in each state

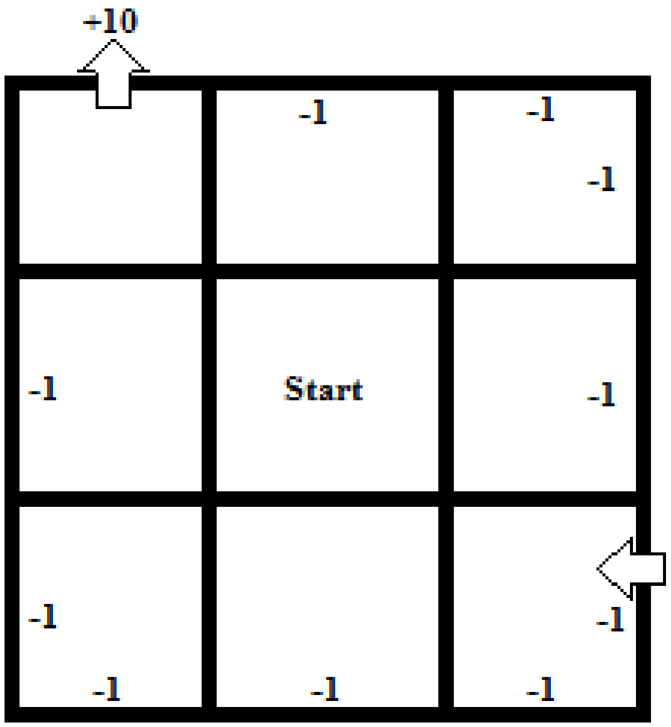


Figure 5.1: The 3 x 3 grid world

(diagonal movement is not allowed). This yields a 9 x 4 Q-table. The 5 x 5 and 9 x 9 grid worlds would have Q-tables of dimension 25 x 4 and 81 x 4, respectively. The Q-table dimensions grow quickly as the environment increases in size. Q-learning updates the Q-table once per time step according to

$$Q_{k+1}(s,a) = Q_k(s,a) + \alpha \left[r_t + \gamma \max_a Q_k(s',a) - Q_k(s,a) \right] \tag{5.2}$$

Notice that it is implicitly assumed that the reward delay is zero because credit for the reward r_t is being assigned to the Q-table entry representing the estimated value of the state-action pair (s,a).

To explore the effects of this implicit assumption, we will first examine what effect a larger delay size has on the agent's performance. We measure performance by recording the agent's average reward throughout the simulation. Next, the average reward results for different delays are normalized using the performance of Q-learning in an undelayed environment. In other words, the normalized average reward $r_{n,i}$ is given by:

$$r_{n,i} = \frac{r_{\text{delay,i}}}{r_{\text{undelayed,i}}} \tag{5.3}$$

where $r_{\text{delay,i}}$ is the average reward obtained by the agent suffering from a reinforcement delay in world i and $r_{\text{undelayed,i}}$ is the average reward obtained by the agent without a reinforcement delay in world i. For example, if the undelayed Q-learning agent achieved an average reward of 2, and the delayed Q-learning agent earned an average reward of only 1, then the normalized average reward is equal to 0.5.

Figure 5.2 shows the relative performance of a Q-learning agent subject to varying mean delays in a 3×3 grid world. The x-axis represents the iteration index of the simulation, which, in this case, ran for 150k time steps. This is enough updates to allow the performance to settle. The agent updated its Q-table once per time step. Observe that larger values of λ, i.e. larger mean and more varied delays, cause the agent's performance to worsen, all else being equal. This is because there is more lag and jitter in the system. Since the agent is using Q-learning, it is assuming that the delay is zero at all times, and so the probability of the agent assigning credit properly is lower when the mean delay is higher.

Next, we will examine the effect that the environment size has on performance. Generally, we expect that performance will worsen when the environment size increases because there is a lower probability that the agent will assign credit correctly by mere luck. When there are more possible state-action pairs, it is less likely that the correct cause of the reward will be selected. In the grid world, there is only one correct cause, and it becomes crowded out by the extra options present in a larger environment.

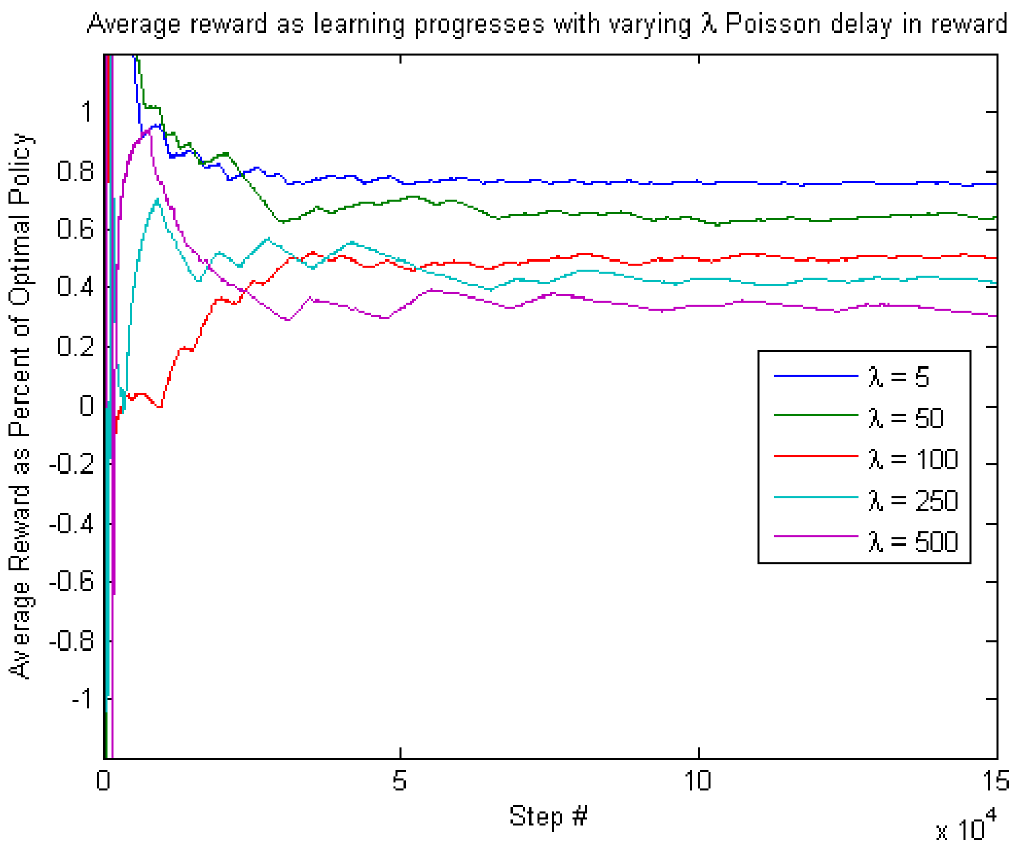


Figure 5.2: 3 x 3 grid world - Effect of increasing the mean delay

To test this notion, simulations were conducted in 3 x 3, 5 x 5, and 9 x 9 grid worlds. Each simulation contains 250k time steps, which is enough to allow the performance to settle. Figures 5.3-5.5 demonstrate that a larger grid world lowers the normalized performance of the agents even when the delays are equal. In each case, the agent in the smaller grid worlds is able to learn more in the same amount of time, although performance is still inferior to that of the undelayed Q-learning agent.

Although delay inhibits performance, as we can see in Figures 5.3-5.5, the normalized average reward does not fall to zero in the 3 x 3 and 5 x 5 grid worlds. This is because the agent can still assign credit properly some of the time by chance, especially in a smaller environment such as these. Some learning takes place as a result, but such luck cannot be counted on in larger environments and is made worse with

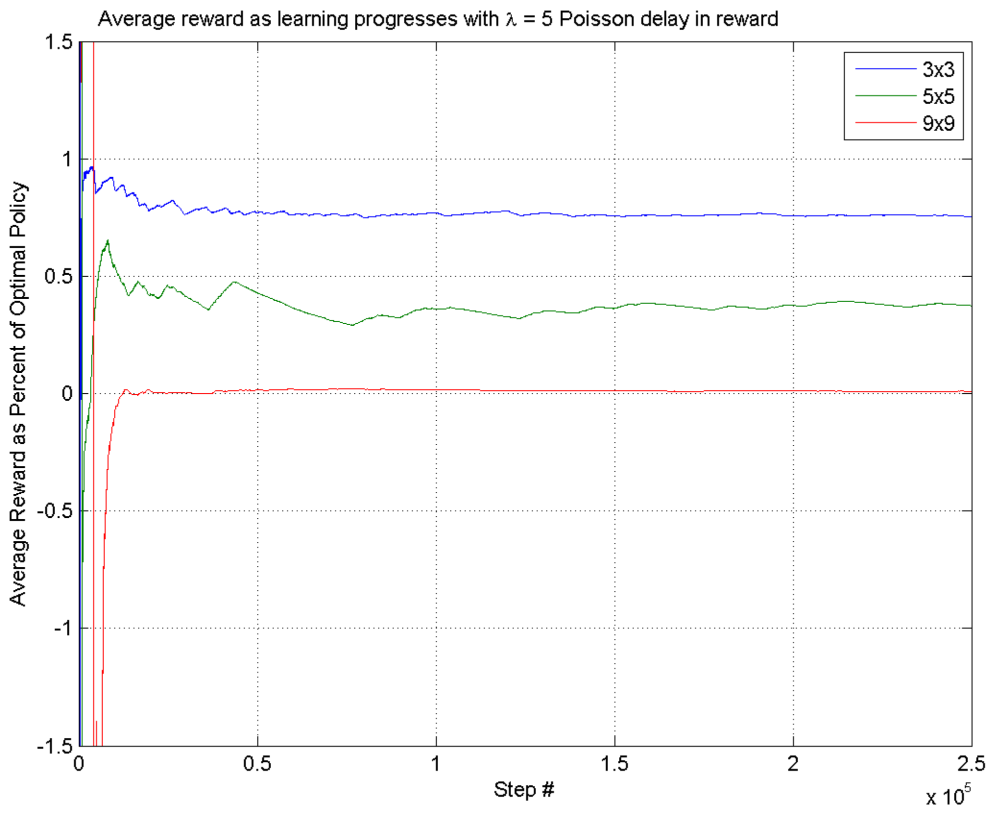


Figure 5.3: Normalized average reward in various environment sizes for $\lambda = 5$

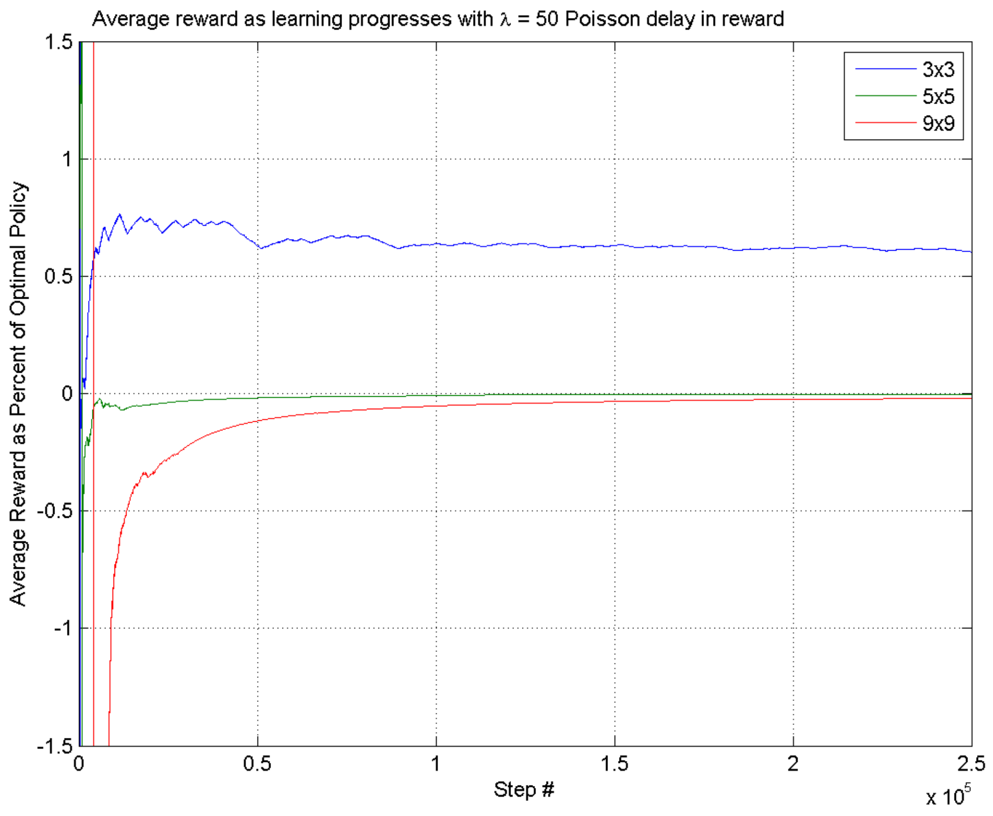


Figure 5.4: Normalized average reward in various environment sizes for $\lambda = 50$

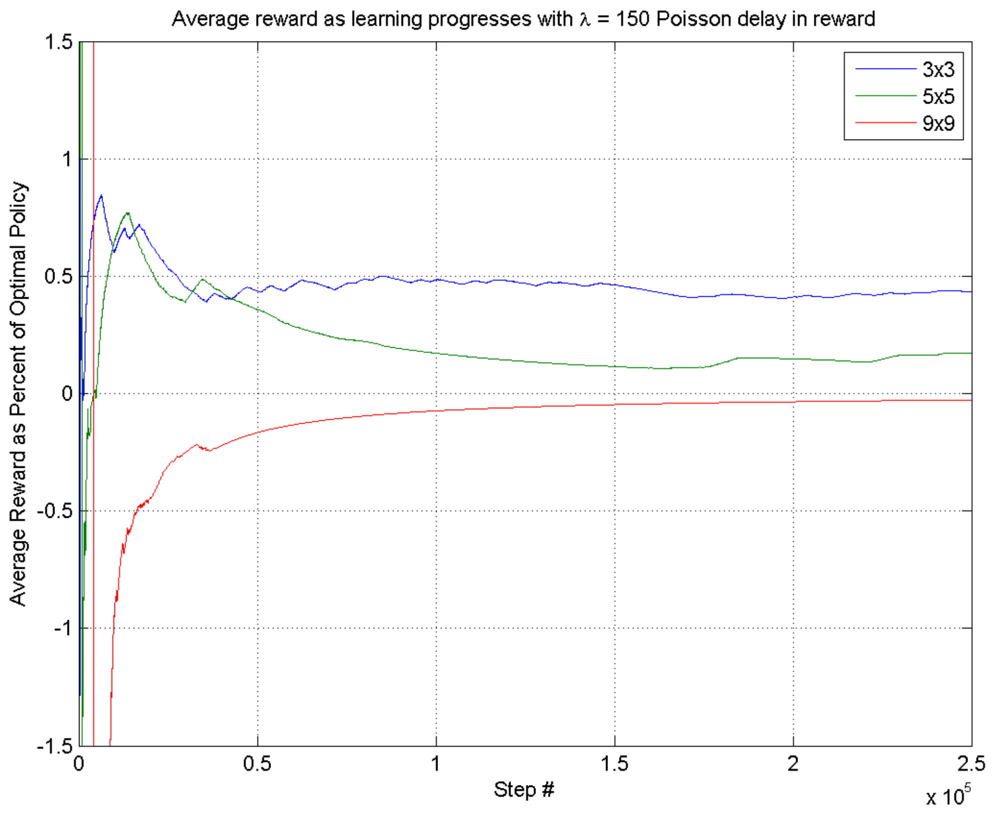


Figure 5.5: Normalized average reward in various environment sizes for $\lambda = 150$

larger mean delays. For example, in the 9 x 9 grid world, the performance fell to 0% of the undelayed performance. This means that the agent almost never reached the reward state at all. Instead, it became lost wandering through the grid world's zero reward states.

From these simulations, we can conclude that the stochastic reinforcement delay problem is not trivial and cannot be solved by using traditional Q-learning. In real-world applications, state spaces are often much larger than the simple grid worlds we study above. If Q-learning is already inhibited in the grid world, then we cannot expect that it will do well without modification when tested in a real application when a stochastic delay is present. For larger delays, it becomes even more troublesome to use Q-learning.

In the above simulations, the agent was having trouble assigning credit properly. What is the best way to remedy these errors? Assume temporarily that the mean reward delay is known beforehand to be λ and the agent decides to attribute any rewards to the actions that are most likely to be responsible. This means that the agent will assume that the cause of any reward is the state-action pair which occurred λ time steps in the past because this is the peak of the Poisson distribution. In other words, the agent maximizes the probability of correct assignment by assuming a constant delay of λ because

$$\arg\max_d \frac{\lambda^d}{d!} e^{-\lambda} = \lambda \tag{5.4}$$

For example, when $\lambda = 5$, the agent should assume that credit belongs to the state-action pair five time steps in the past.

When this is assumed, Q-learning performs significantly better because credit is assigned properly more frequently. Figure 5.6 shows the performance of 31 Q-learning agents. Each agent assumes that the delay is $\lambda + i$ where i is the estimate error in units of time steps. For example, if we assume that the mean delay is $\hat{\lambda} = 27$ when it is actually $\lambda = 17$, this corresponds to a delay estimate error of $i = 10$ on the x-axis. We measure the normalized performance of each of the 31 agents and plot them against their delay estimate error. Each agent was given 150k iterations to learn. The reported performance in Figure 5.6 is the average of these ten epochs for each agent. The true mean delay was $\lambda = 17$ and the simulation was done in the 9 x 9 grid world. The peak of the curve has a slight performance dip, and this is likely due to the stochastic nature of the exercise.

Notice that the performance curve takes on the shape of a Poisson distribution approximately centred around the true mean delay, λ. When the agent assumes that the mean delay is $\lambda + i$, the estimate strays farther from the most probable state-action cause and the agent's performance tends to worsen. The estimates which are closest to the truth reach a performance which is 60% of the optimal, undelayed performance.

Although this method of assuming a constant delay produces fairly good, albeit sub-optimal performance, we cannot necessarily assume that the agent is aware of the time delay statistics beforehand. Therefore, we design multiple model Q-learning in a way that it can learn about the delay statistics on-line.

5.1.1 Multiple Model Q-Learning in the Grid World

Here we will compare the performance of multiple model Q-learning to that of single model Q-learning. Comparisons are conducted in the 9×9 grid world test bed, as shown in Figure 5.7. We will number the states as in Figure 5.8 for easy reference.

To facilitate more learning than in Q-learning during simulation, the highest feasible delay estimate parameter in multiple model Q-learning, $\hat{\lambda}_{\text{max}}$ is set to 50 time steps. The true value, λ, is set to 13, although the agent does not know this beforehand. Thus, this maximum estimate is chosen so that we can be fairly sure that the

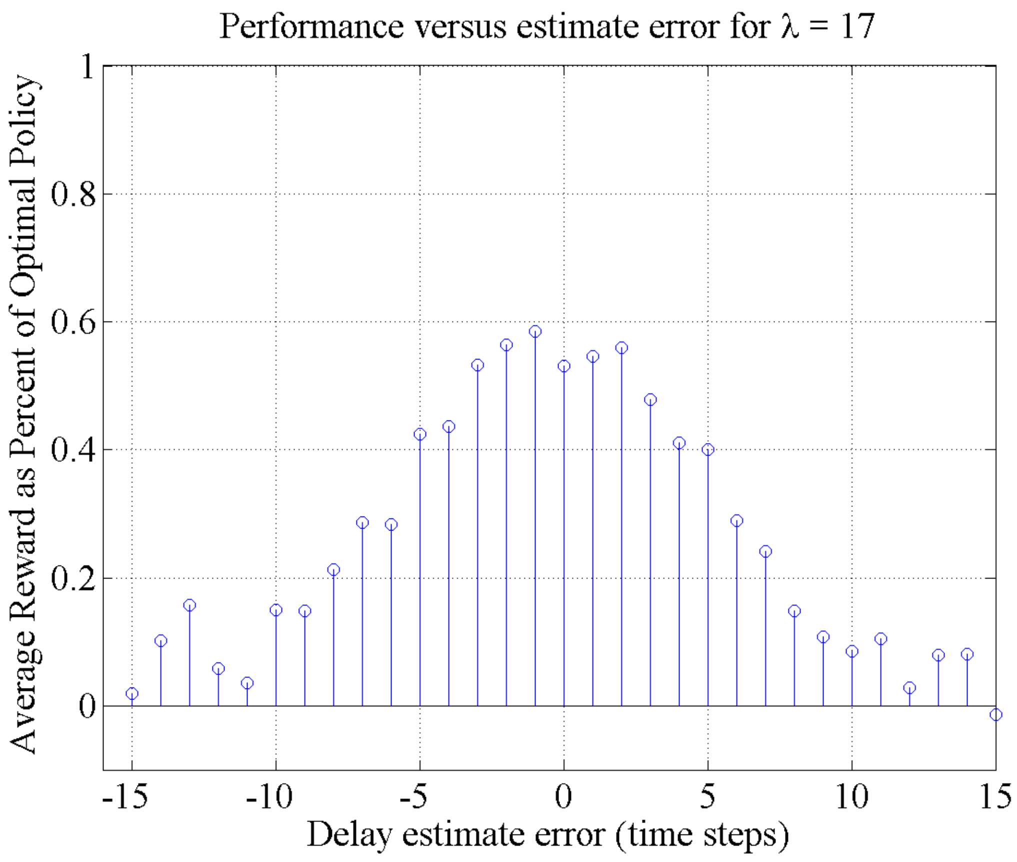


Figure 5.6: 9 x 9 grid world - Performance after 150k iterations for errors in the delay estimate $i = \hat{\lambda} - \lambda$

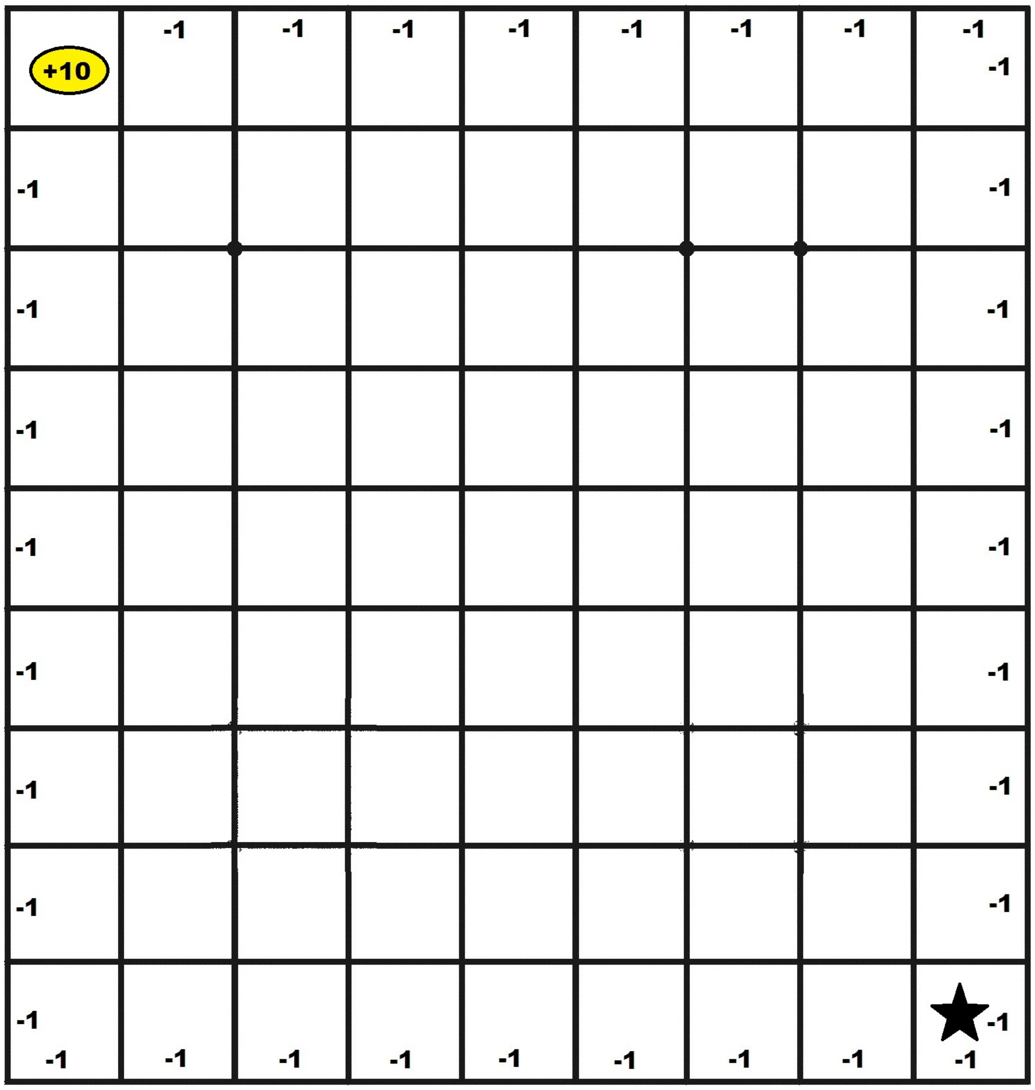
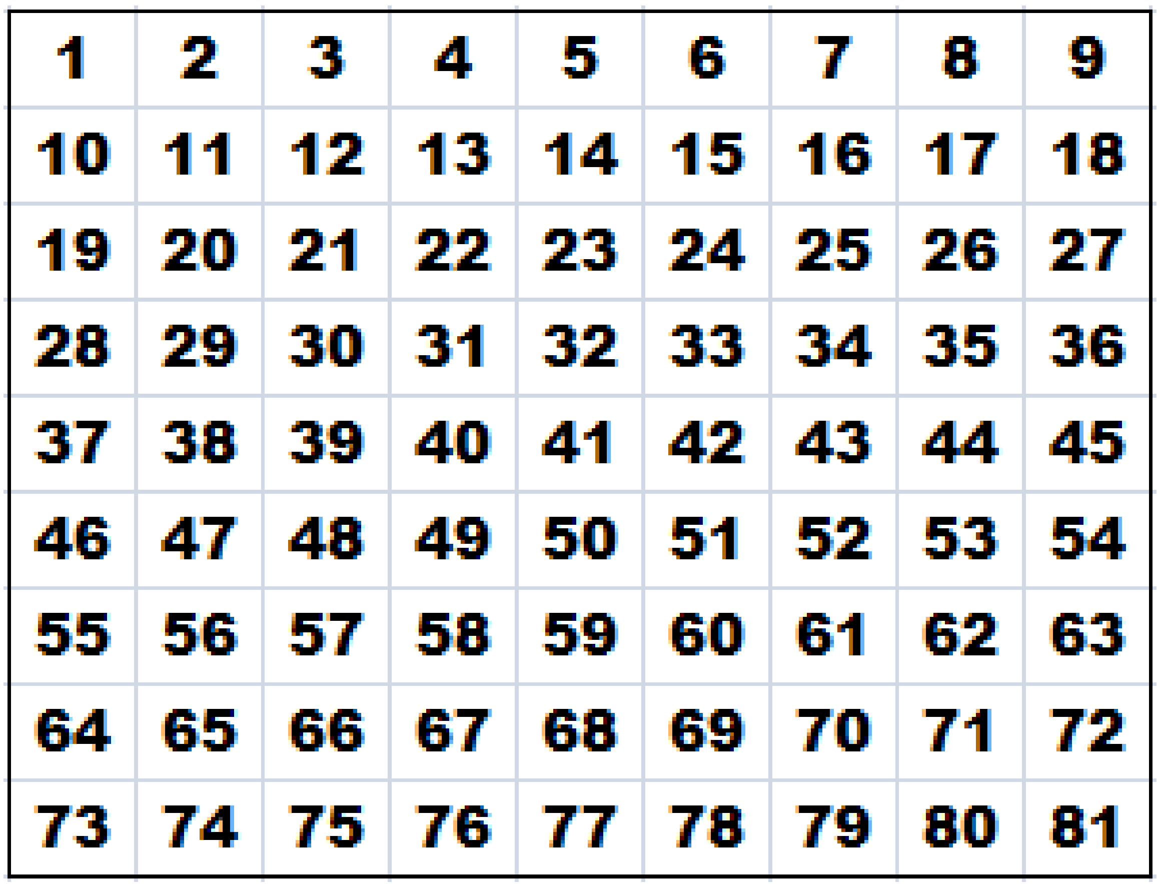


Figure 5.7: The 9 x 9 grid world



1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18
19	20	21	22	23	24	25	26	27
28	29	30	31	32	33	34	35	36
37	38	39	40	41	42	43	44	45
46	47	48	49	50	51	52	53	54
55	56	57	58	59	60	61	62	63
64	65	66	67	68	69	70	71	72
73	74	75	76	77	78	79	80	81

Figure 5.8: The 9 x 9 grid world - state numbering

true delay is significantly less than the agent's maximum estimate. The agent does not know the true delay beforehand, and so the maximum must be set in a way that is likely to exceed the true value. This defines the maximum number of parallel estimate models which must be maintained. For example, if the true delay were $\lambda = 20$, but we chose a maximum estimate of $\hat{\lambda}_{\max} = 15$, then it would not be possible for the agent to consider $\lambda = 20$ as a possible delay, although it could come fairly close.

The grid world has 81 states, 4 actions per state, and we allow 50 estimate models. The multiple model Q-table therefore has dimension 81 x 4 x 50. The table should be thought of as 50 parallel Q-tables of size 81 x 4. Each of the 50 parallel tables has an implicit assumption about the time delay. For example, the first table assumes that the time delay is *always* zero. The second table, assumes that the delay is *always* one, etc.

Figure 5.9 shows the improved performance from using the multiple model Q-learning algorithm in a 9×9 grid world. The simulation was run for one million time steps. The multiple model Q-learning method achieves a near-optimal performance, although it is not perfect because of the forced exploration chance, ϵ. The undelayed Q-learning policy is able to reach the reward state in 17 time steps, as does the multiple model Q-learning policy, but sometimes it is forced to explore. Why is the optimal policy such that it takes 17 time steps to reach the reward state and return to the start? This is the number of actions required to move from the bottom-right corner to the top-left corner and collect the reward in the 9 x 9 grid world if diagonal moves are not allowed. In comparison, single model Q-learning under delay has difficulty finding the reward state from the delayed reward signal. Instead, it learned to eventually avoid the walls (-1 reward), but did so by following a loop in the centre of the grid world (0 reward).

As the agent explores and learns, it is generating a policy. When learning is finished, the agent has travelled through the states and explored the environment,

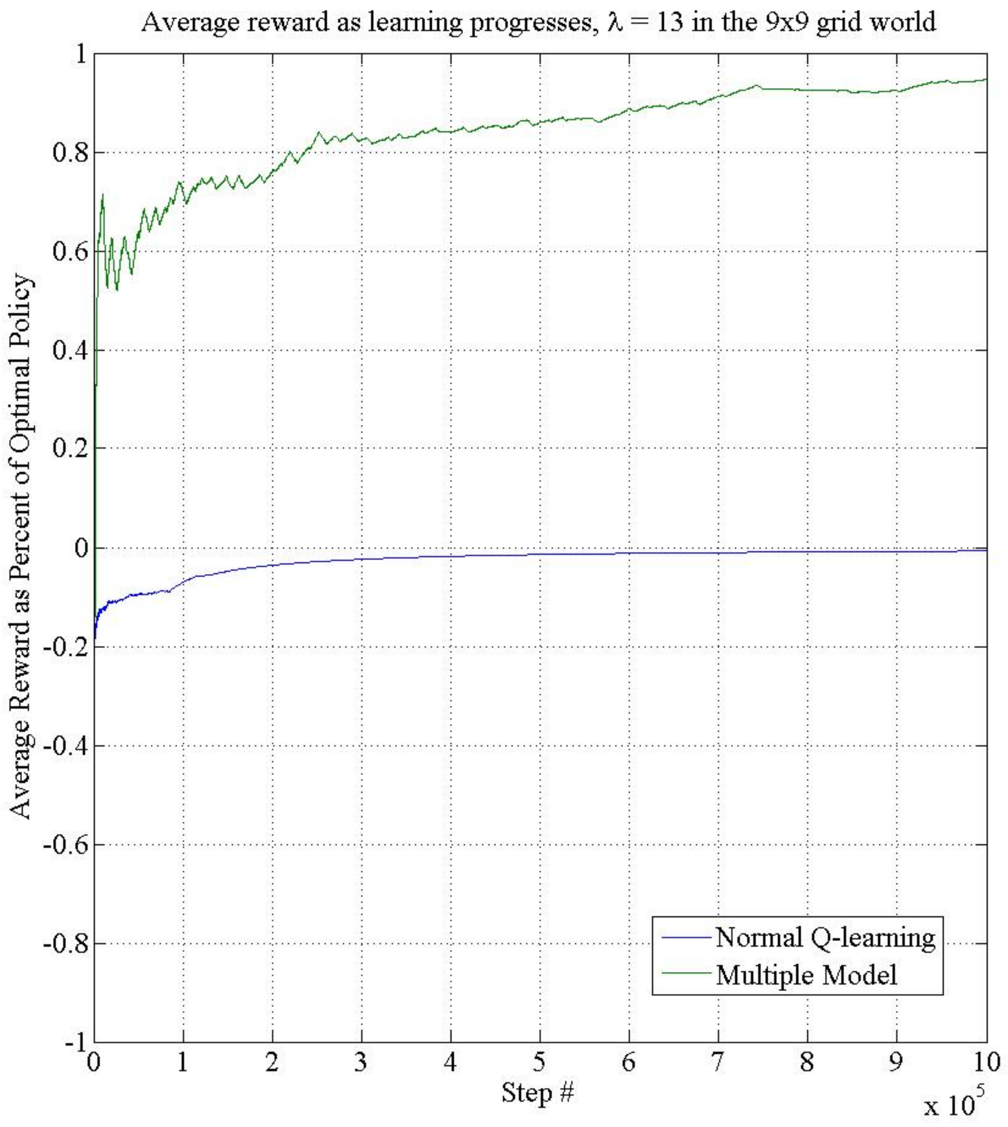


Figure 5.9: Relative performance of multiple model Q-learning vs single model Q-learning in the 9×9 grid world with random delays present

as in Figure 5.10. This figure shows how often the agent visited each state during learning. We can see that it visited the states along the diagonal most frequently, as this is the shortest path to the reward state which also avoids from the penalizing walls.

If a deterministic control policy is formed using the maximally valuable entries in the Q-table, the learning agent will behave as in Figure 5.11. In other words, for every state s we choose the most valuable action as $a^* = \arg\max_a Q(s, a, \hat{\lambda})$. After the multiple model Q-tables have converged, the value of $\hat{\lambda}$ no longer matters because all 50 parallel Q-tables contain the same values for the states and actions which are used in the final policy. In practice, the values are similar, but not exactly equal. Even the table which assumes that there is no delay eventually converges to be equal to the table with a delay estimate which was most accurate, for the states that are used in the final solution. The Q-values of the seventeen state-action pairs used in the deterministic policy are shown in Table 5.1. Note that the action in state one does not matter because all actions lead to state 81 and receive the same reward of +10. The mean values and variances in this table are taken across all delay estimate $\hat{\lambda}$. That is, for each state-action pair, we calculate the mean and variance of $Q(s, a, \hat{\lambda})$ across all $\hat{\lambda}$.

An important point is that when Q-learning uses a Q-table with a zero delay estimate in isolation, it will produce a poor policy due to the stochastic reward delay, but when the Q-table is placed in parallel with other Q-tables with differing time delay estimates, the learning will be spread across the tables. The act of updating all models in parallel at each time step allows the different estimates to share their learning with one another until the best estimate dominates the others and the delay estimate value is no longer important. Formally,

$$|Q_k(s, a^*, \hat{\lambda}) - Q_k(s, a^*, \hat{\lambda}')| \to 0 \text{ as } k \to \infty \qquad \forall \hat{\lambda}, s \in S^* \tag{5.5}$$

Table 5.1: Multiple model Q-values along the optimal route after 10^6 iterations

State	**Action**	**Mean**	**Variance**
1	3	21.38	0.72
2	4	21.35	0.73
11	1	21.38	0.73
12	4	21.36	0.72
21	1	21.34	0.71
22	4	21.30	0.72
23	4	21.26	0.73
32	1	21.27	0.78
33	4	21.41	0.79
34	4	21.49	0.79
43	1	21.55	0.79
52	1	21.57	0.79
61	1	21.55	0.79
62	4	21.54	0.79
71	1	21.53	0.79
72	4	21.53	0.73
81	1	21.43	0.72

where S^* is the set of all states s which are in the optimal path used by the agent and k is the time step index. For example, if we need to travel through the states 1, 2, and 11, to behave optimally, then these are members of S^*. This means that the entries relevant to the optimal path all converge to the same fixed point, for all states along the optimal path, despite their different delay estimates, $\hat{\lambda}$.

Note that in Figure 5.11, the policy is optimal, but the normalized average reward reported in Figure 5.9 is not equal to 1. During learning, the exploration rate is still non-zero and this forces the agent to act suboptimally some of the time. After learning is complete, this is no longer true, and the agent does behave optimally because the exploration rate is now zero.

Intuitively, multiple model Q-learning spreads the rewards out across all states which are responsible for them. In the end, this causes the states in the optimal path to have approximately the same value.

If the agent were to begin in state 81 (9,9), then it would move roughly along the diagonal to reach the reward state. The optimal path consists of taking some combination of the actions *up* and *left* to reach the reward state directly. Moving along the diagonal also reduces the risk of bumping into a wall. Through the states most visited during learning, the preferred actions tend to be *up* (light blue) and *left* (pink). Similarly, Figure 5.12 shows the probabilistic policy formed from the same Q-table, where the agent will move in each direction with a given probability based on the current state, as described by the color scale. Warmer colors indicate a higher probability of taking that action in that state.

We will now compare multiple model Q-learning to the performance of single model Q-learning. As seen in Figure 5.9, by using the traditional Q-learning algorithm, the agent does not achieve a positive reward, meaning that the agent has not learned to reach the goal state. As well, Figure 5.13 shows that during learning, single model Q-learning explored poorly and learned to loop around the centre of the

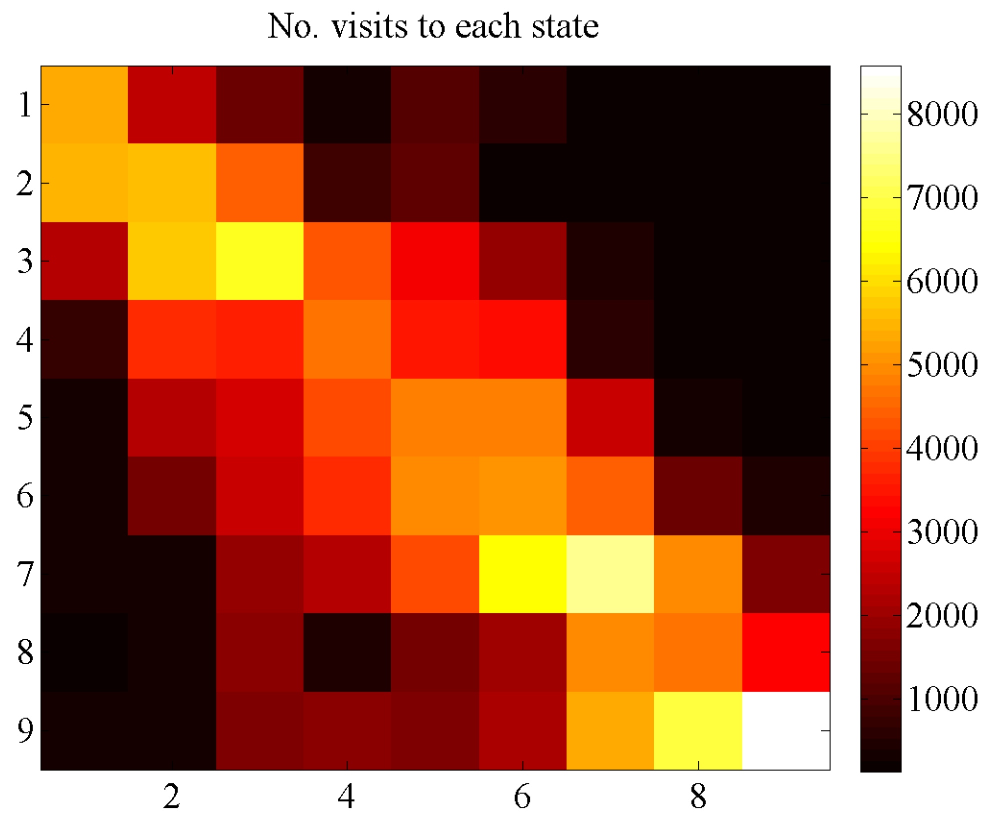


Figure 5.10: How often each state is visited during learning when using multiple model Q-learning.

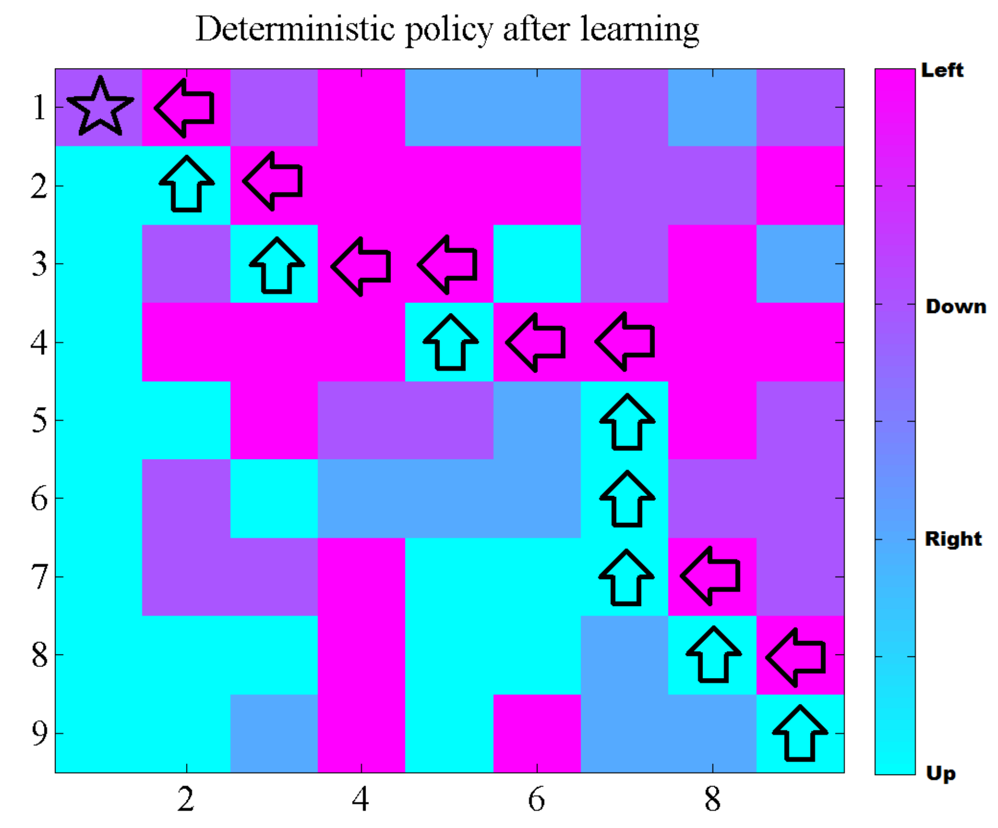


Figure 5.11: Deterministic policy formed using multiple model Q-learning

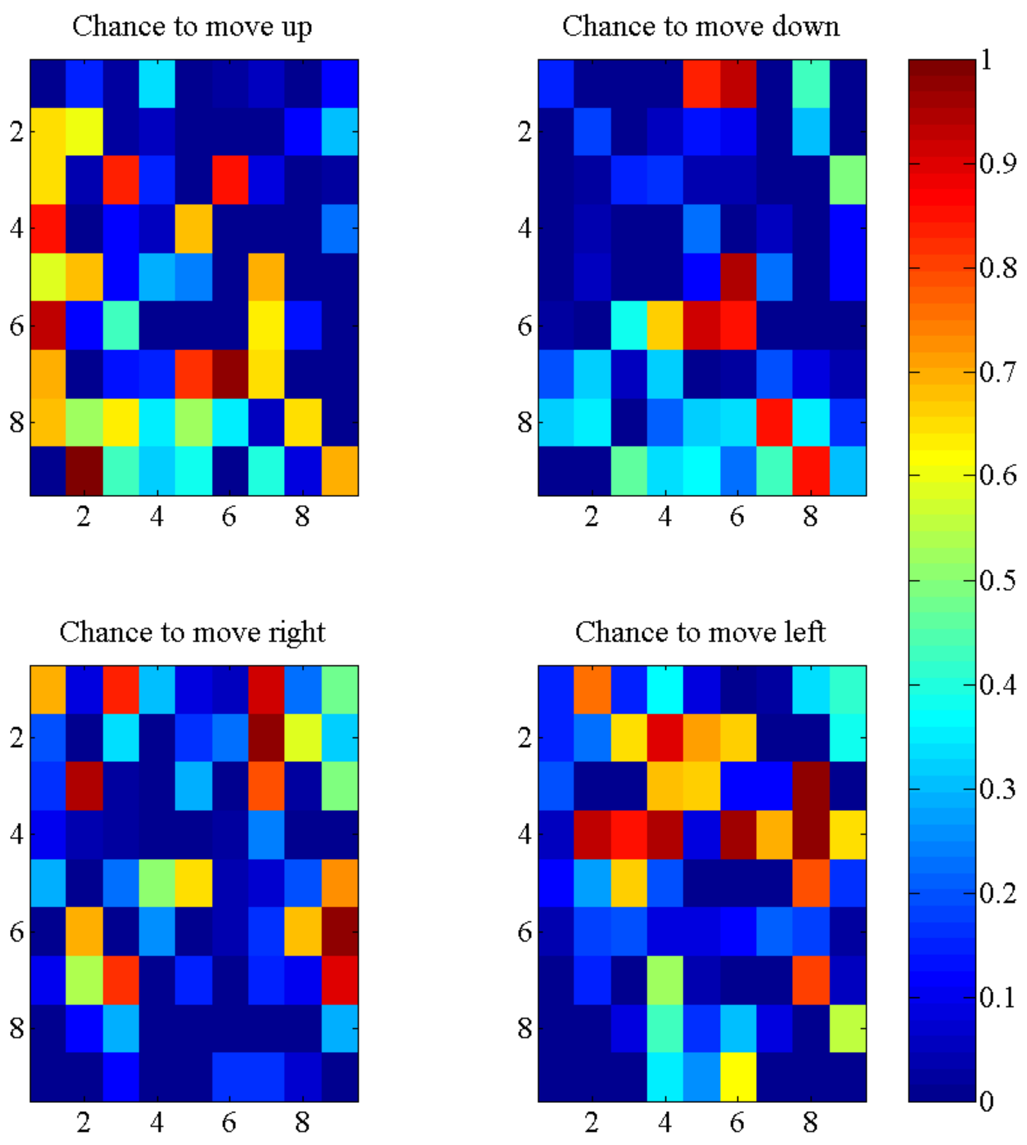


Figure 5.12: Probabilistic policy formed using multiple model Q-learning

grid. It may be that it has learned to avoid the walls at least, but the goal state is never reached. This stuck result was consistent across multiple runs, and one example is reported here. The agent's Q-table was initialized optimistically, initialized using ϵ-greedy, and set to initialized to zero, but in all of these cases, the Q-learning agent became stuck.

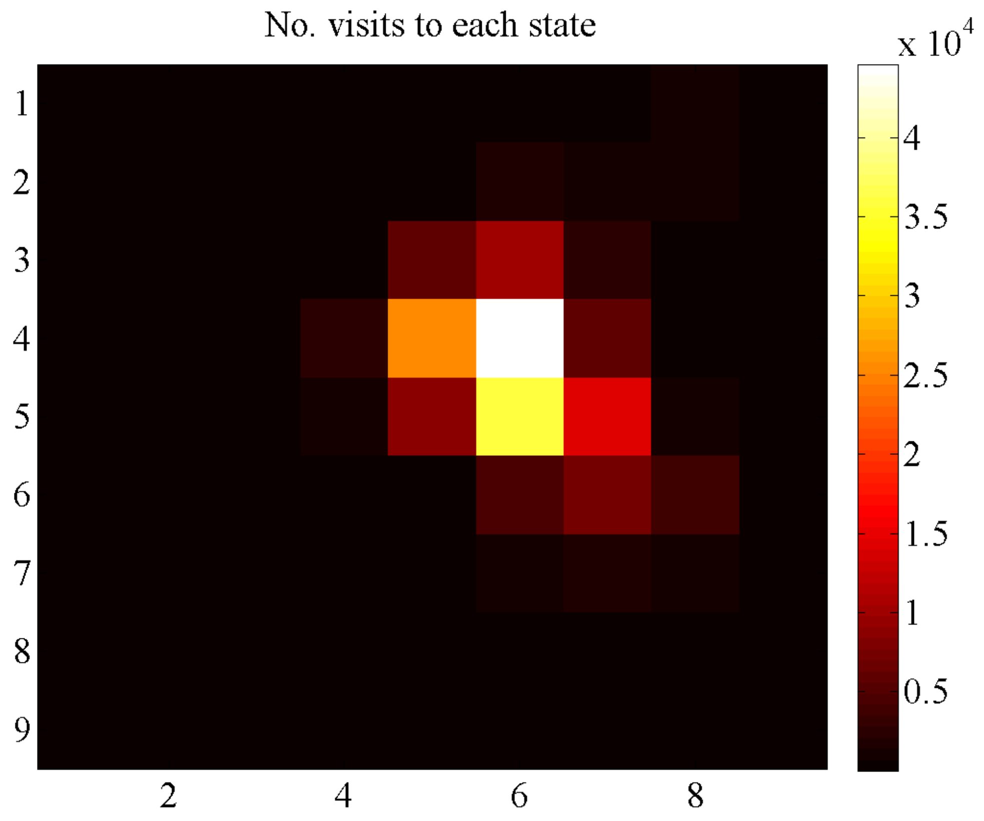


Figure 5.13: How often each state visited during learning when using single model Q-learning

Q-learning's deterministic policy is shown in Figure 5.14. This policy is generated in the same way as that of multiple model Q-learning, by selecting the most valuable action for each state. No clear path emerges which would move the agent from bottom-right to the reward state at top-left. Instead, the agent becomes stuck in a loop in the zero reward region, but manages to avoid the walls (-1 reward).

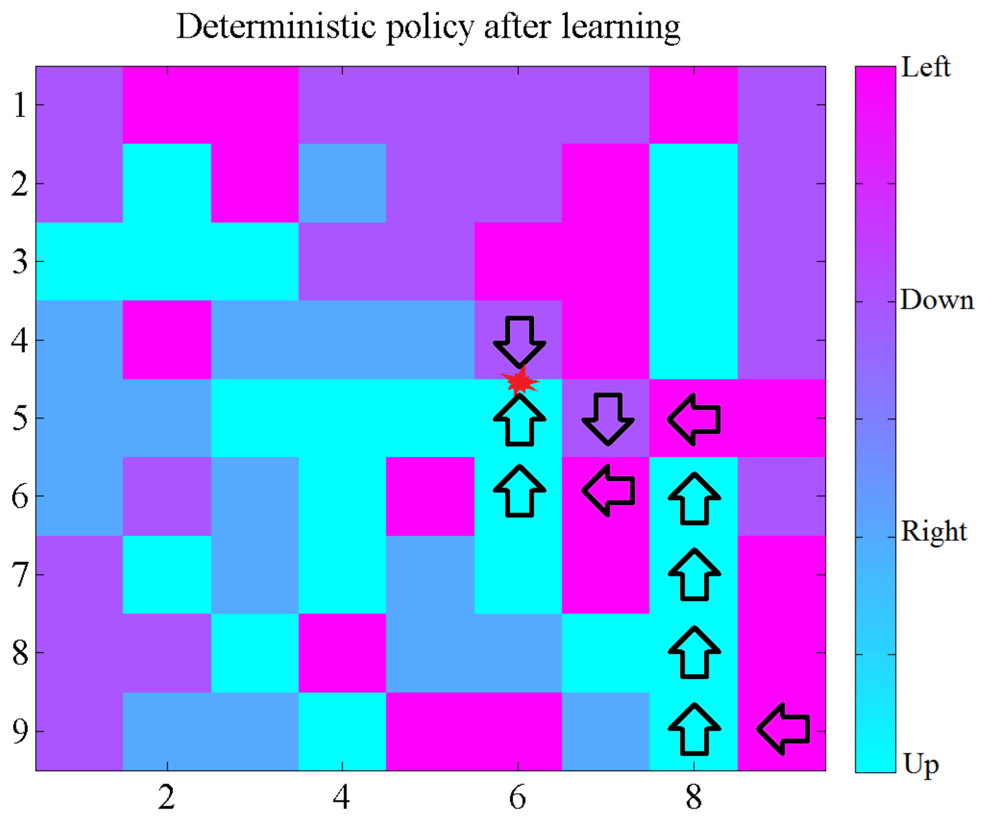


Figure 5.14: Deterministic policy formed from using single model Q-learning

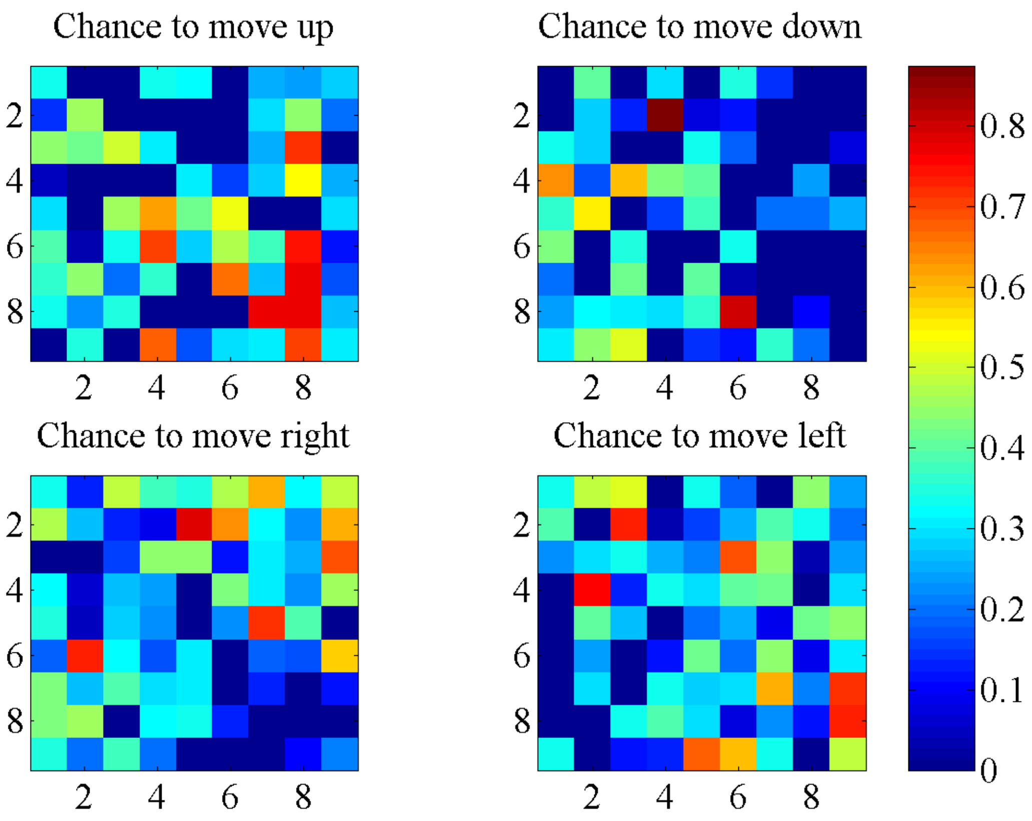


Figure 5.15: Probabilistic policy formed from using single model Q-learning

Figure 5.15 is the probabilistic policy formed from Q-learning. The poor exploration leads to a policy of mostly random behaviour because most of the states have not been visited extensively. Note that warmer colors indicate more decisive behavior.

Running both policies for 1,000 iterations after learning is complete, the multiple model Q-learning control policy achieves an average reward of 0.59 (+10 in 17 steps) and the Q-learning policy achieves an average reward of 0.00 (it becomes stuck in a loop). An optimal policy would achieve an average reward of +10 in 17 steps, or 0.59. These results demonstrate that the novel method facilitates more learning than the Q-learning method.

So far, we have shown that, in the grid world at least, multiple-model Q-learning outperforms single-model Q-learning significantly. Multiple model Q-learning explores its environment more thoroughly, learns more from the experience that it gathers from the exploration, and ultimately, better maximizes its reward function by producing better behaviour.

5.2 Webots simulation - line-following robot

Now that we have explored the problem in the grid world, we will move to the Cyberbotics' Webots software interfaced with MATLAB. This software is used to simulate robots in real world environments, complete with simulated physics and realistic tasks. The software is interfaced with MATLAB for data management and observation. In our particular application, we simulate a single e-puck mobile robot. Its goal is to follow a racetrack as quickly as possible while being centred on the line without any prior knowledge of the shape of the track. This is called a robot line-following task and is analogous to a car driving along a road with the goal of remaining centred on a solid line.

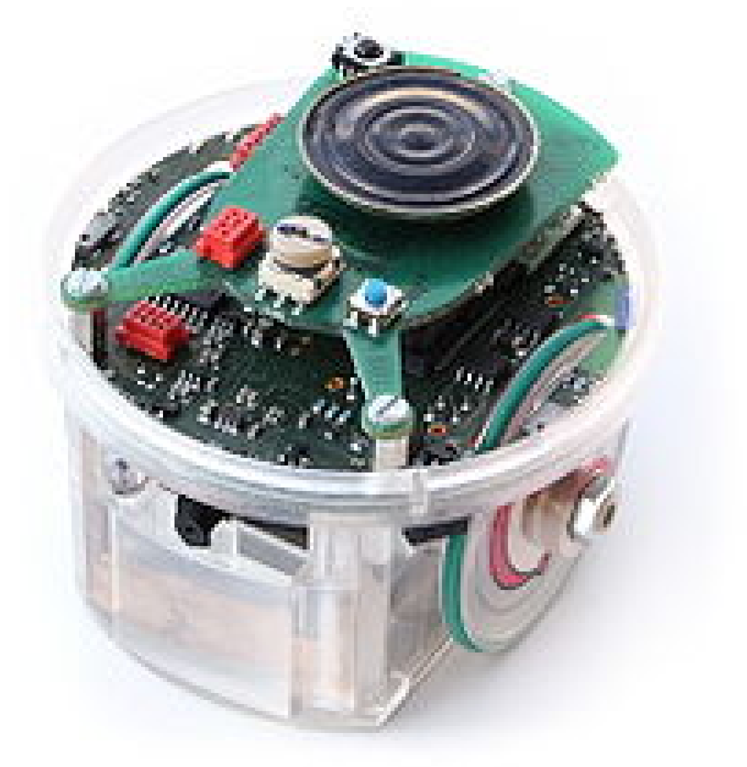


Figure 5.16: e-Puck Mobile Robot

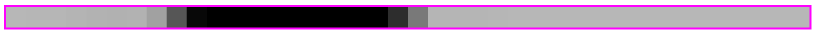


Figure 5.17: e-Puck Camera View

5.2.1 Problem Overview

In this application, we would like a small e-Puck mobile robot (Figure 5.16) to drive along a racetrack. Our goal is to make the robot drive quickly while remaining centred on the line. As an example, Figure 5.18 contains one possible racetrack and includes the e-Puck placed on top for a comparison of size.

The robot uses its front camera to look down and detect the line just in front of its body. The camera is initialized to see a rectangle 1 pixel high by 40 pixels wide, as in Figure 5.17. The dark pixels represent where the line has been detected and the light pixels represent where the line is not visible.

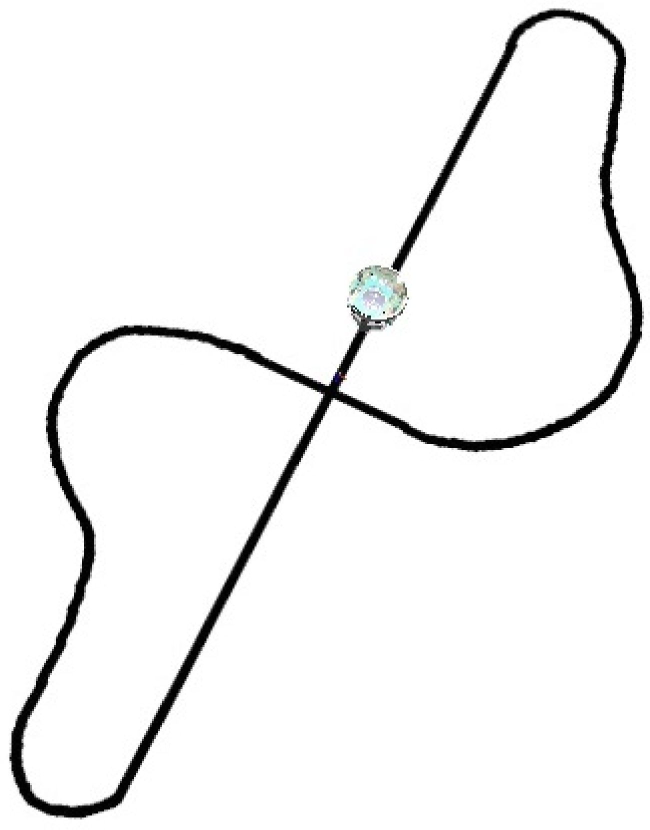


Figure 5.18: An Example Racetrack

5.2.2 State and Action Spaces

The robot's state s is defined based on the centre of mass of the line in its camera view. To do this, each pixel is considered either to contain the line (1) or not (0) and the centre of mass m of the line, as observed by the camera, is calculated and rounded to the nearest integer as in Equation (5.6). The function $f(i)$ is a binary function which indicates whether or not the line is present in pixel i, i.e. $f(i_{\text{seen}}) = 1$ and $f(i_{\text{unseen}}) = 0$. The resulting index $1 \leq m \leq 40$, as calculated in Equation (5.6), is the robot's state and refers to the pixel closest to the centre of mass of the visible line.

$$m = \text{round}\left(\frac{1}{40}\sum_{i=1}^{40} f(i)i\right) \tag{5.6}$$

For example, if the line is only visible in pixels 18 to 23, the centre of mass is 20, rounded from 20.5. The robot's state is therefore $s = 20$.

To define a discrete action space, for the e-puck, which is a differential wheeled robot, the velocity of each wheel is restricted to an arbitrary resolution. Differential wheeled robots can independently adjust the speed of each wheel. In this thesis, the wheel speed is restricted to a resolution of 100 with a maximum speed of 300 on either wheel. Therefore, the action space contains 7 actions as in Table 5.2. This quantization is chosen so that the number of actions is just high enough to make it possible for the robot to navigate the track, while being kept as simple as possible. In particular, if the number of speed resolution is too low, then the robot will not be able to turn smoothly on turns with various sharpness. This resolution was found to be adequate for the experimental task at hand.

The experiment's variable delay is set to be Poissonian with mean and variance $\lambda = 90$. The delay is intentionally chosen to be a large value to demonstrate the robustness of multiple model Q-learning. We must assume that the designer does

Table 5.2: e-Puck Action Space

Action	**Left Wheel Speed**	**Right Wheel Speed**
1	0	300
2	100	300
3	200	300
4	300	300
5	300	200
6	300	100
7	300	0

not know this value beforehand. Somewhat arbitrarily, we set $\hat{\lambda}_{max} = 150$ as the designer's estimate of the maximum feasible delay in this system. This estimate must safely exceed the true mean time delay, which we do not know beforehand.

The resulting table $Q(s, a, \hat{\lambda})$ has the dimensions $40 \times 7 \times 151$. Note that the delay estimate dimension (151) can be updated entirely in parallel if the processing hardware allows it, because these calculations are independent of one another.

Now that the state and action spaces are defined, we must construct a suitable reward function to express our objectives to the robot. Our goal is for the robot to move as quickly as possible while keeping the line centred in its view. If the line is not visible to the robot at all, the robot is punished and receives a reward of $r = -1$. If the line is visible, we construct the reward as:

$$r = \frac{v_{\text{left}} + v_{\text{right}}}{2v_{\text{max}}} \left(1 - \frac{|p_{\text{line}} - p_{\text{centre}}|}{p_{\text{centre}}}\right) \tag{5.7}$$

where v_{left} and v_{right} are the speeds of the left and right wheels, respectively, v_{max} is the maximum wheel speed (300). A wheel speed of 300 means that the wheels will make three full rotation every ten seconds. Next, p_{line} and p_{centre} are the pixels corresponding to the centre of the line and the centre of the camera, respectively. In particular, we set $p_{\text{centre}} = 20$, because this is the centre pixel of the camera.

The maximum reward is achieved when wheel speed is high and the line's centre is aligned with the camera's centre. Although the maximum reward is 1 in this function, the robot may receive rewards which overlap and amount to more than 1. However, we do assume that the rewards are bounded. In practice, it is impossible to consistently receive rewards of +1 because the robot cannot travel at full speed forward when navigating the turns on the racetrack. It must slow one of its wheels to turn. As a result, the maximum practical average reward is between 0.8 and 0.9.

5.2.3 Comparing adaptive and traditional techniques

To provide a baseline for comparison, a simple proportional controller was implemented to make the robot follow the track. We test the controller with and without stochastic feedback delays to make the case for the use of adaptive learning techniques at all in this application. If a simple controller can solve the problem optimally, then there is no need to implement a more complicated reinforcement learning technique which we must train and which will perform suboptimally until learning is complete.

In this comparison, the performance of the proportional controllers is measured in terms of the average reward received because this is how the performance of the reinforcement learning schemes is typically measured as well.

The proportional controller is designed to measure the difference between the line's centre and the cameras centre, Δ. We then set the speed of each wheel v_1 and v_2 with the gain set empirically to $k_p = 6$:

$$v_1 = 300 - k_p|\Delta| + k_p\Delta \tag{5.8}$$

$$v_2 = 300 - k_p|\Delta| - k_p\Delta \tag{5.9}$$

When no delay is present, the proportional controller performs well, as expected, as we see in Figure 5.19. The controller's final average reward is approximately 0.9 after 1700 time steps, which is practically perfect for the application. The e-Puck follows the track nearly perfectly, centring itself on the line and moving swiftly. In this undelayed controller, the reward is still measured and reported with a Poissonian time delay of $\lambda = 90$, although the controller can sense the line without delay. Of course, the controller does not take the reward signal into account in any way, so it does not affect its performance.

Next, we introduce a Poissonian delay of $\lambda = 90$ time steps into the camera's observation of the line. This would be properly classified as an observation (state) delay, but the proportional controller does not use a reward function. Delaying the state observation, which is the only input to the reward function, is the closest analogy to a reward delay for a proportional controller. The delay is present in the value of Δ. At time step t, the value of Δ which is observed is delayed by a random amount of time steps with mean and variance λ. In other words, at time step t, we expect to be using our observation from time step $t - 90$. When this delay is present, the proportional controller becomes unstable and drives wildly about the track, missing the line most of the time and randomly finding the line a small part of the time. The average reward of this controller is very low, at -0.6 after 1700 time steps, as shown in Figure 5.19.

Now that we have set a high bar and a low bar, we compare multiple model Q-learning and single model Q-learning in the same scenario. Both the multiple model and single model Q-learners were thoroughly trained in the environment and then the resulting Q-tables were used at the beginning of the 1700 step run seen in Figure 5.19. This was necessary to place the four competitors on an even footing because the reinforcement learners take far longer to reach a steady-state performance than a proportional controller because they are adaptively learning and exploring the environment while the proportional controller is set beforehand rather than being adaptively tuned.

Observe, in Figure 5.19, that Q-learning does a better job than the delayed proportional controller because it has some limited capacity to learn about its environment, although it does not perform very well. It reaches an average reward of around 0.3.

The performance of multiple model Q-learning was the best in the delayed case, with an average reward of just over 0.6. This is the closest performance to the near-perfect case of the undelayed proportional controller.

The main goal of this comparison is to demonstrate that in applications as noisy as this scenario, a controller with fixed values, such as the simple proportional controller in this example, or other PID controllers, etc. cannot function because of the massive lag in feedback. This is the main reason that we are proposing a novel learning algorithm. Adaptive learning algorithms such as Q-learning are much more able to handle massive noise in feedback in the present experiment than a deterministic optimal control scheme.

5.2.4 Comparing different learning techniques

We will now compare single model Q-learning to multiple model Q-learning more closely, having already provided evidence that they are suitable techniques for this type of problem, as opposed to a more traditional type of control. To make the comparison, the Webots e-Puck was trained using single model Q-learning and then using multiple model Q-learning. In both cases, a mean time delay of $\lambda = 90$ was present in the reward signal and the Q-table was trained for 150k time steps. Figure 5.20 compares the performance of the two algorithms. Observe that multiple model Q-learning was able to learn more effectively, achieving an average reward of about 0.5 versus an average reward of only 0.1 for Q-learning. Intuitively, this means that multiple model Q-learning was able to learn more from the same information, and in the same amount of time. Thus, we have achieved better performance from the same

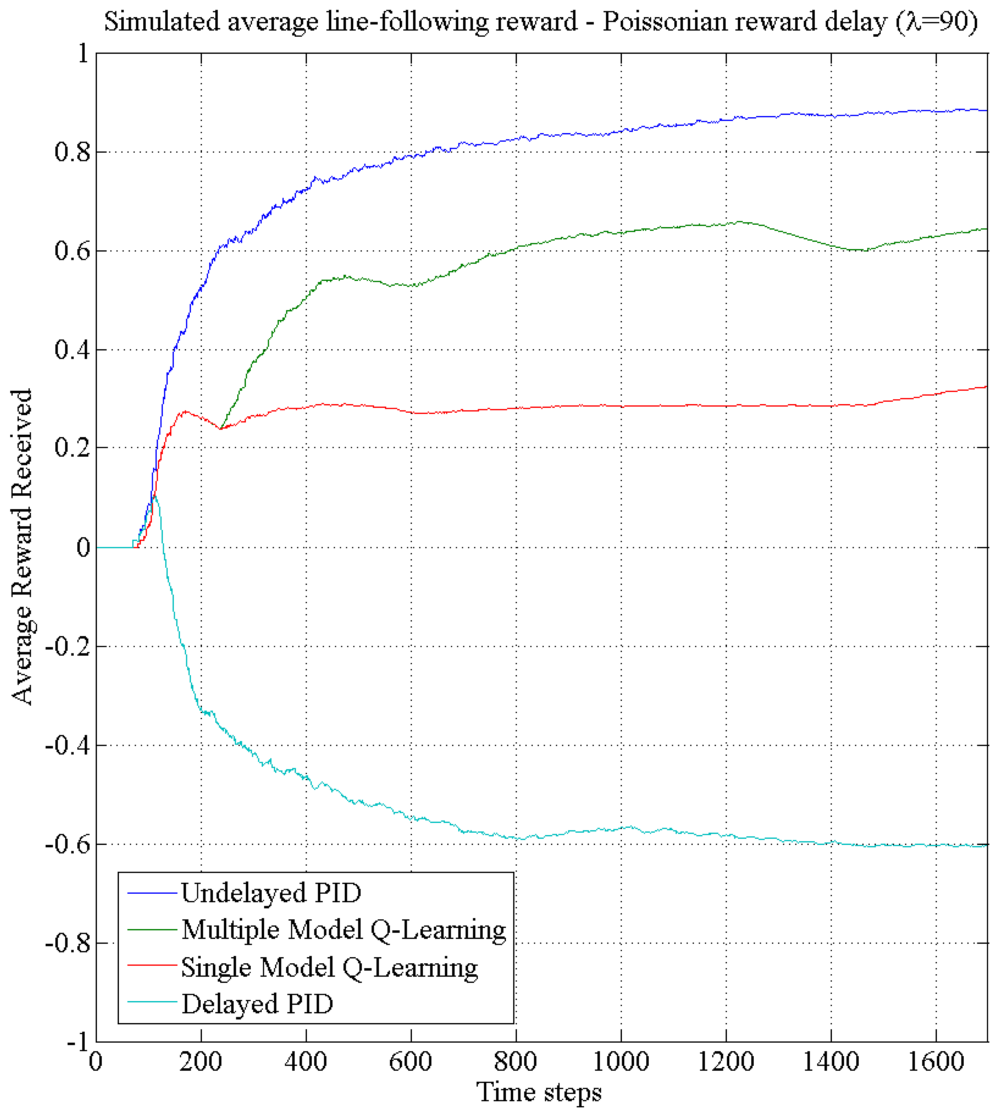


Figure 5.19: Webots - Controllers' Performance Under Delay vs. Undelayed PID

learning opportunity. Put another way, we can see in Figure 5.21 that the accumulated error of multiple model Q-learning is much less than that of single model Q-learning.

Because of the stochastic nature of the algorithms, the following result is a five-run average to support the previous single run results. Five runs were found to be enough to lower the standard deviation of the results to a reasonable amount. In Figure 5.22, which is a result averaged over five runs with standard deviation indicated, note that multiple model Q-learning significantly outperforms single model Q-learning in the line-following task. The former receives an 5-run average reward of about 0.55 while the latter achieves only 0.12. The standard deviation is slightly lower for multiple model Q-learning as well. Seen from another perspective, in Figure 5.23, the novel approach accumulates significantly less error than does Q-learning. The 5-run average accumulated error of multiple model Q-learning is only about 40% of that of single model Q-learning, even when given the same amount of time to learn. The accumulated error is the combined sum of all the $|\Delta|$ terms across all time steps, which represents the distance of the line's centre from the camera's centre during each time step. In other words, the robot is consistently able to track the line more closely, which is the designer-specified objective.

The improved performance is important because real-world experience is usually expensive in machine learning applications. If the agent can be made to learn more from less, it can learn more efficiently and effectively. The term expensive may refer to actual cost, or it may refer to difficulty in providing the experience. For example, in the e-Puck experiments, it was necessary to manually pick up the robot when it became lost and reset it on the track, which was a bothersome and time consuming activity.

This section has provided evidence to show that multiple model Q-learning can outperform single model Q-learning in a legitimate application, despite the presence of hefty stochastic reinforcement signal delays.

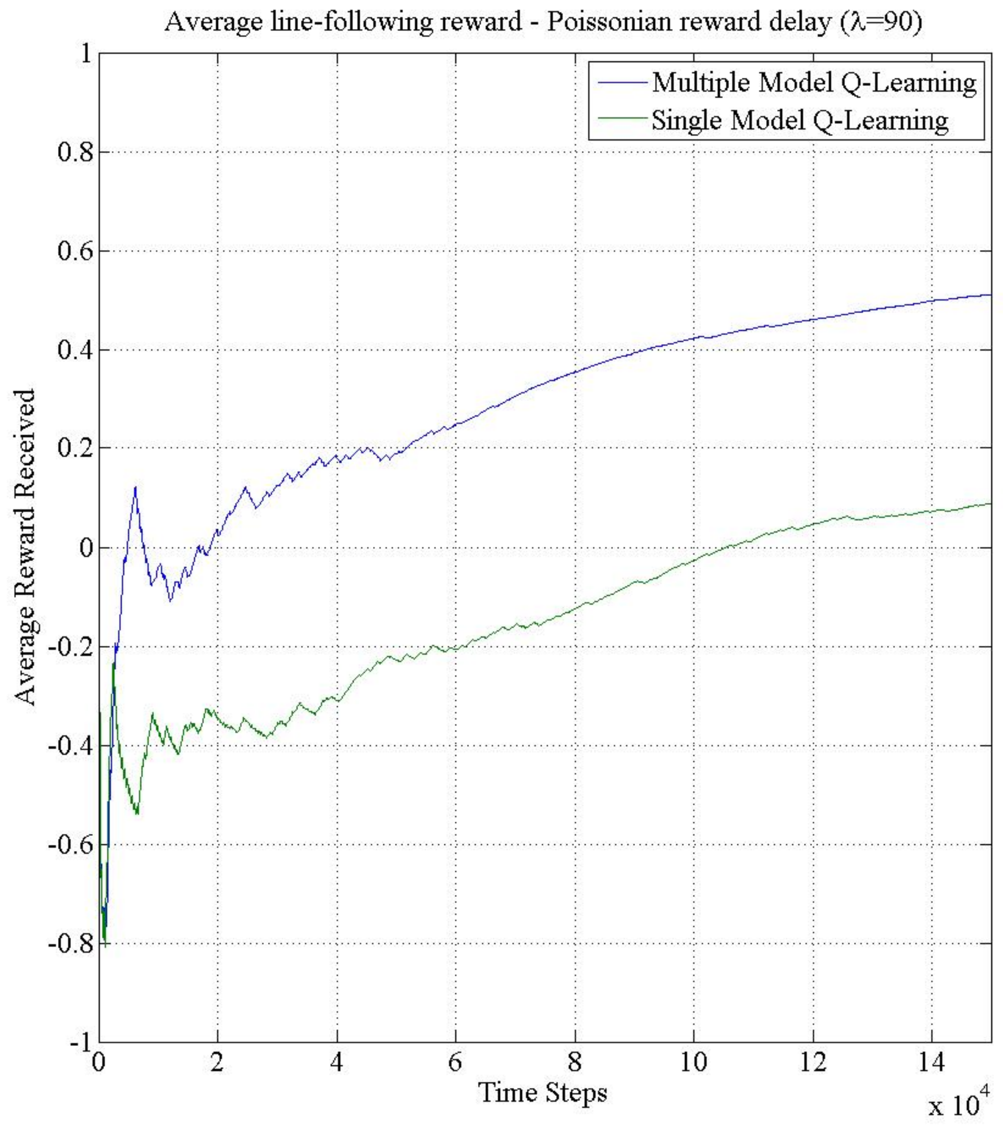


Figure 5.20: Webots - (single run) average reward during learning ($\lambda = 90$)

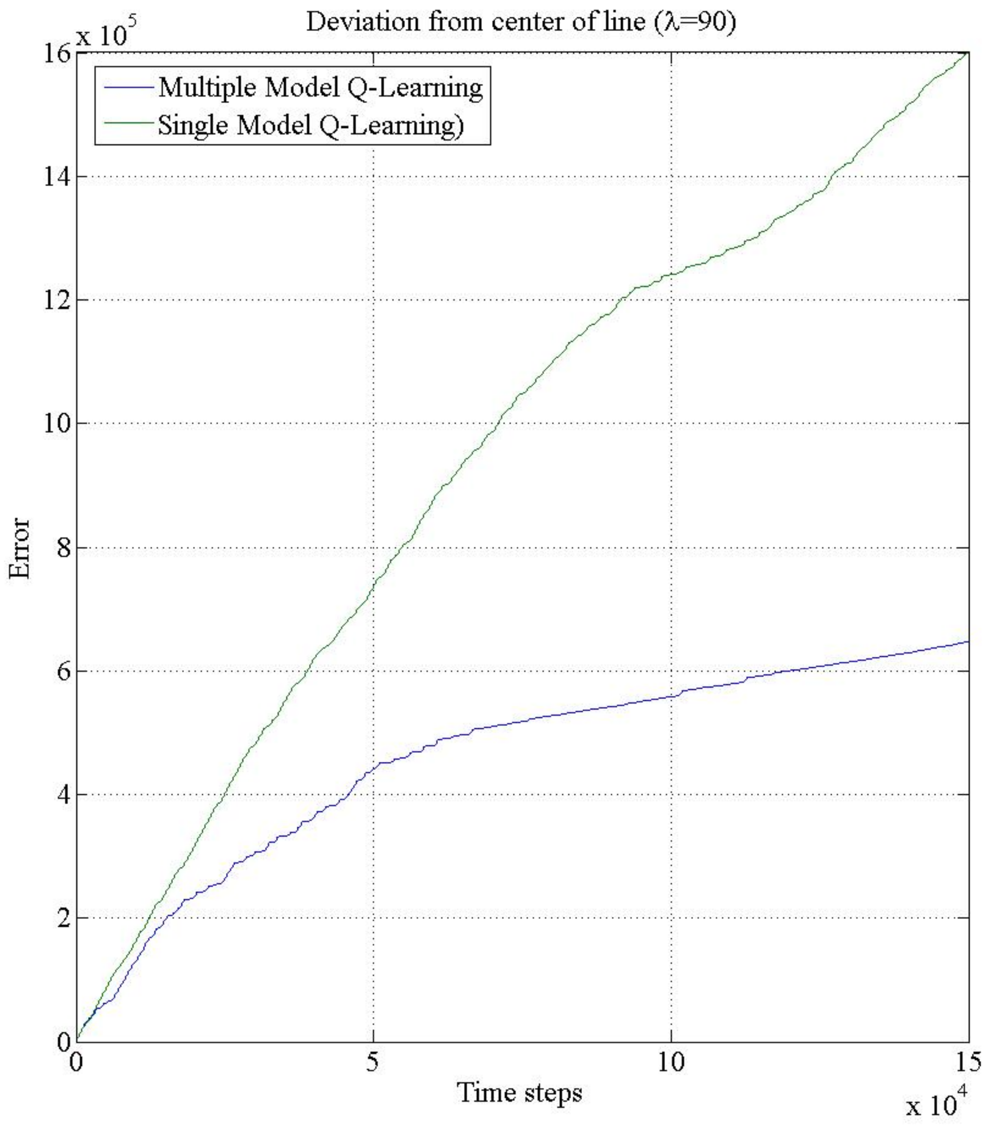


Figure 5.21: Webots - (single run) accumulated error during learning ($\lambda = 90$)

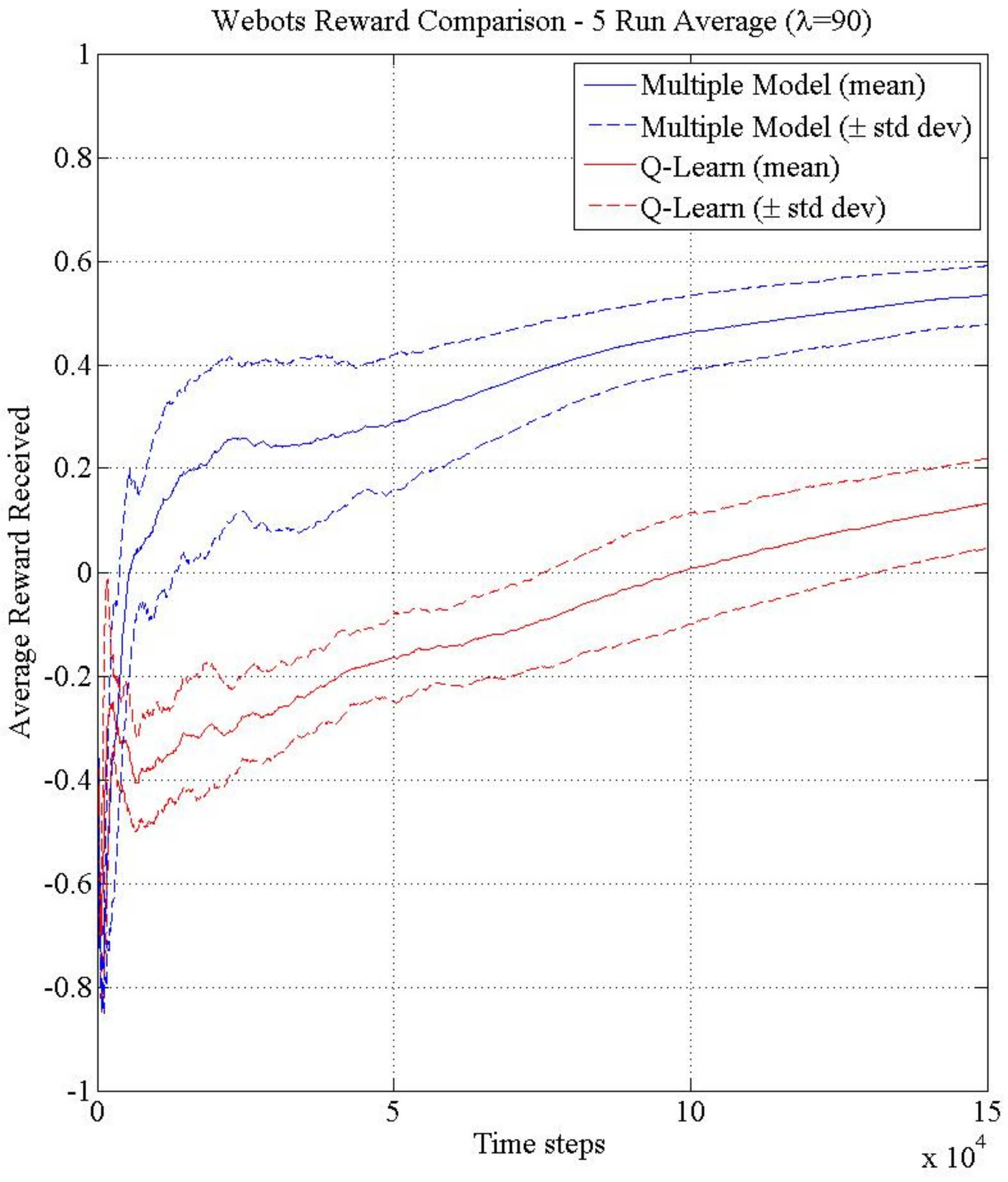


Figure 5.22: Webots - (5 runs) average reward during learning ($\lambda = 90$)

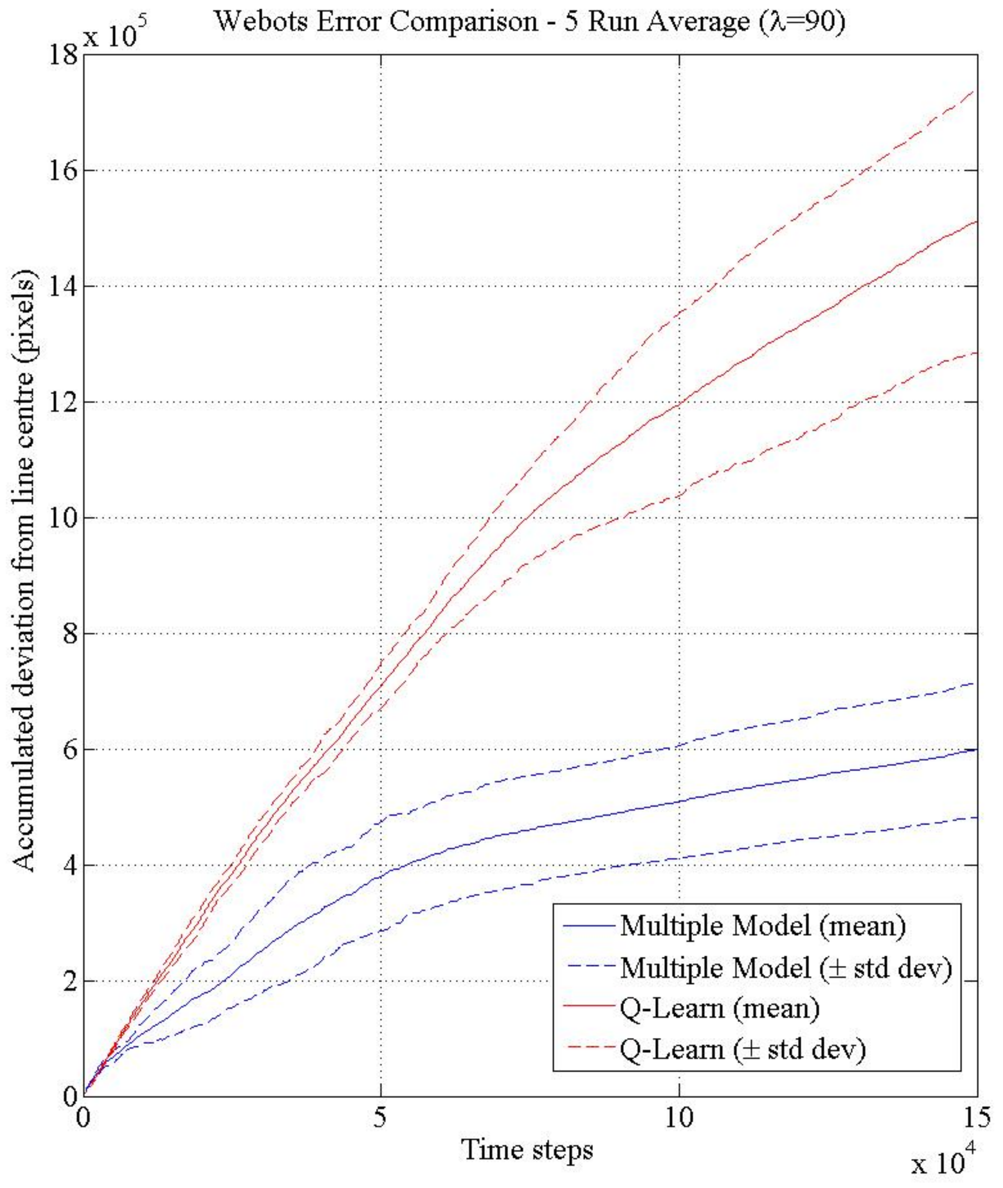


Figure 5.23: Webots - (5 runs) accumulated error during learning ($\lambda = 90$)

5.3 Experimental Results

This section will further the demonstration of the performance of multiple model Q-learning by testing the algorithm in a real experimental set-up of the simulations discussed in the previous section.

The experiments were performed on a scale printout of the Webots racetracks using an e-Puck mobile robot. The e-Pucks have the same sensors and reward function as in the simulations. The robots received commands through a Bluetooth connection with Webots linked with MATLAB. The mean reward delay was set to $\lambda = 90$ time steps with one time step being approximately 200 ms.

We start with a comparison of single model Q-learning and multiple model Q-learning. Single-model Q-learning was compared to multiple model Q-learning in the presence of a delay, paralleling the comparison in the Webots simulation. Both agents received prior training in simulation for 500,000 time steps because of the need for automatic resets when the robot loses the track for too long. When training was finished, the robots were placed on the track and allowed to run for 1500 time steps. The agents were allowed to continue updating their Q-tables during the experiments in the same way as during the simulation because the simulation is not a perfect approximation of the experimental environment.

Figures 5.24 and 5.26 compare the average reward obtained for single model Q-learning and multiple model Q-learning. Single model Q-learning inherently assumes that there is no reward delay which corresponds to the first model of the multiple model Q-learning ($\hat{\lambda} = 0$). As a reminder, the reward scheme allows for a minimum average reward of -1 if the line is always missing and a maximum average reward of +1 if the robot is moving the maximum speed exactly on the centre of the line, which is not practically possible. The practical maximum reward is around that achieved by the undelayed PID during simulation (0.9) in Figure 5.19 because the track has long turns and the robot cannot move full-speed forward while turning. By definition, the average reward is a useful measure of performance because a higher reward means that the robot is better achieving its objectives.

In the first run, multiple model Q-learning achieved an average reward of just over 0.2, while single model Q-learning obtained an average reward of only -0.2. In the second run, we saw results of just under 0.2 and just under -0.3, respectively. In both cases, the two algorithms performed worse in experiment than they did in simulation, which we would expect because of additional difficulties present in the real world which are neglected in simulations. For example, the real world has wheel slippage on paper while the Webots simulation does not. Nevertheless, these experiments demonstrate that, over time, multiple model Q-learning allows the robot to learn more and thus perform better than single model Q-learning.

Put differently, Figures 5.25 and 5.27 show the error accumulated by each agent throughout a run where the error is defined as the distance of the line's centre from pixel 20, the centre of the camera. Observe that single model Q-learning accumulates error more quickly than the multiple model approach. In the first run, single-model Q-learning accumulated about 70% more error than the multiple model method. In the second run, single model Q-learning accumulated about 55% more error than its multiple model counterpart.

Thus, multiple model Q-learning outperforms single model Q-learning in terms of allowing the e-Puck to achieve the goals of the designer, which is to drive quickly while being centred on the line when a Poissonian reward delay is present. The additional models facilitate more learning in the agent because they allow several hypotheses to be entertained at once. The useful estimates tend to make better decisions and override the poor estimates. In single model Q-learning, there is only one inherent assumption about the time delay – that it is zero.

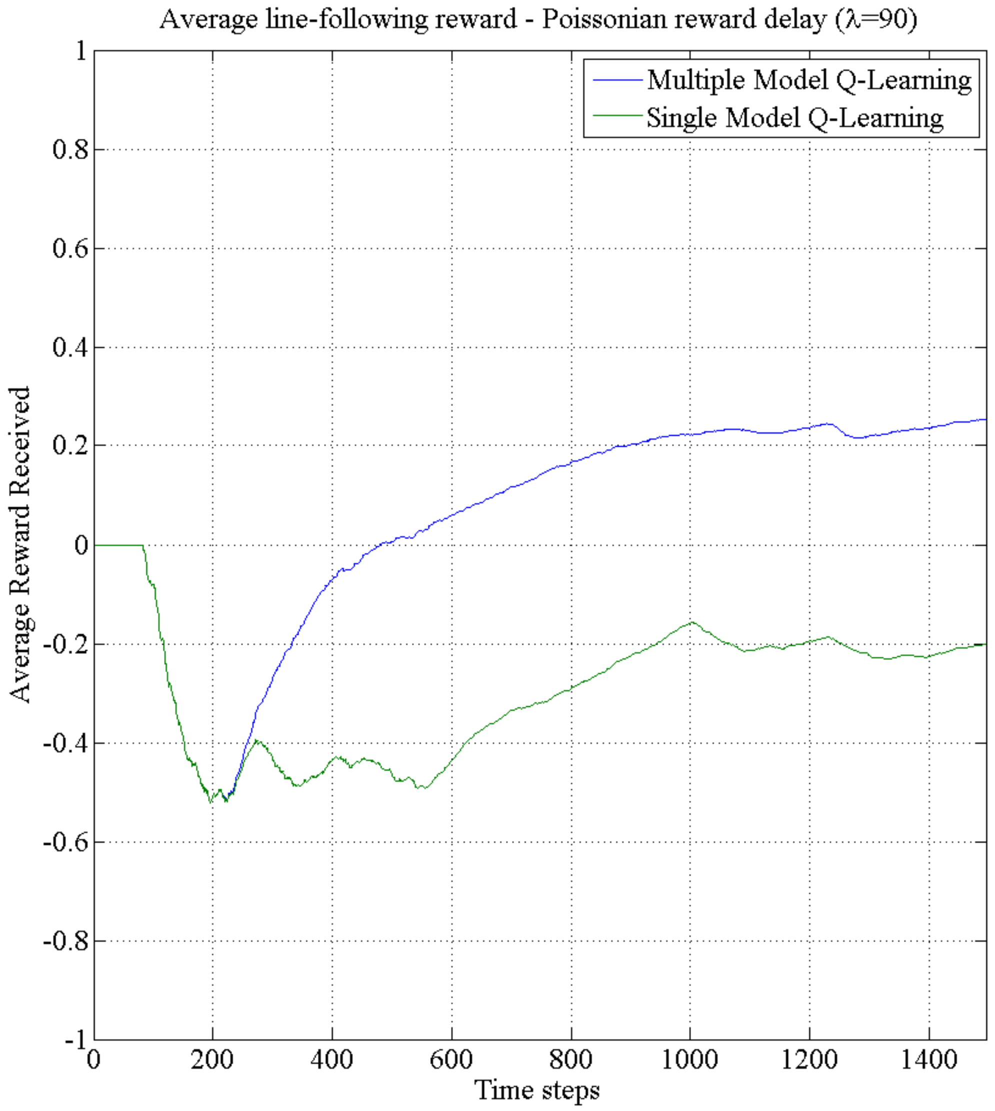


Figure 5.24: Comparison of Average Reward During Experiment 1

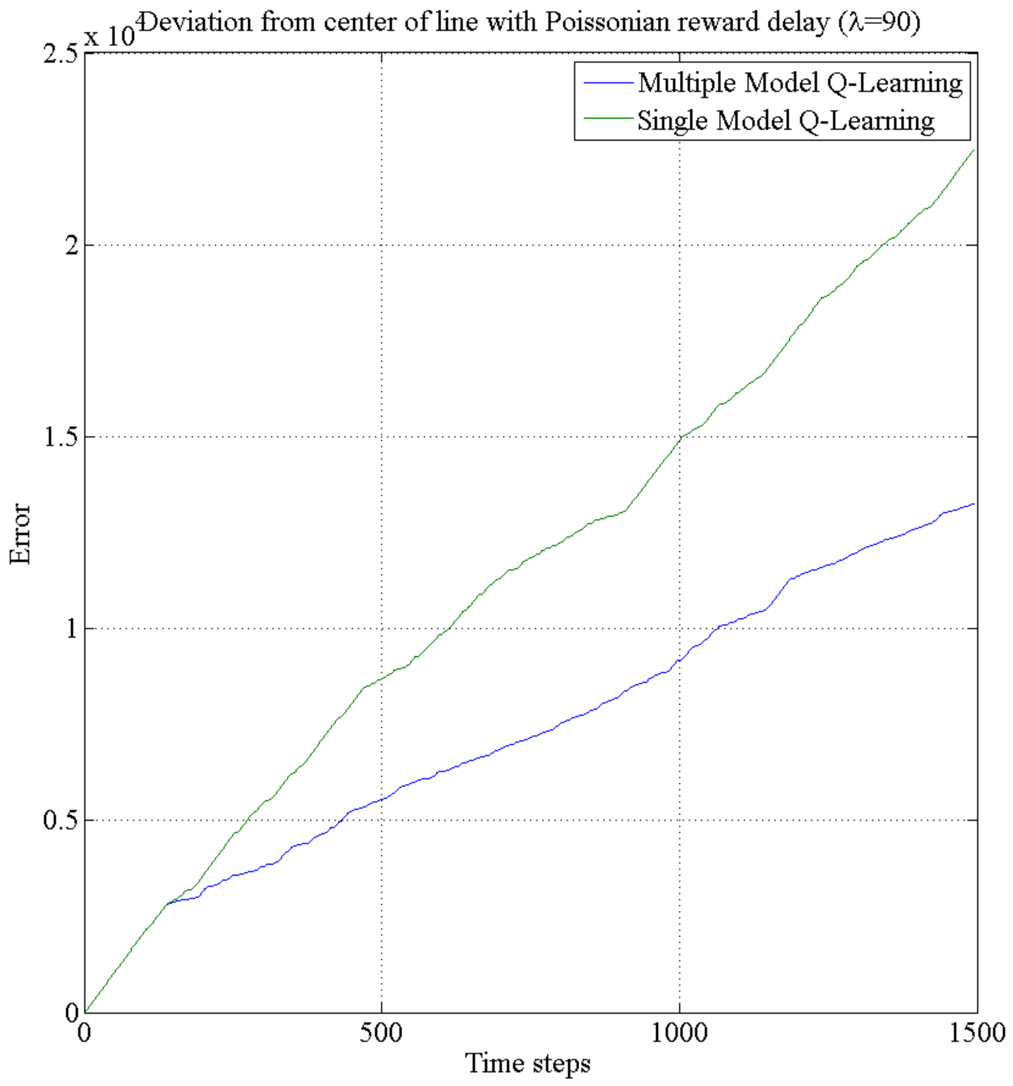


Figure 5.25: Comparison of Accumulated Error During Experiment 1

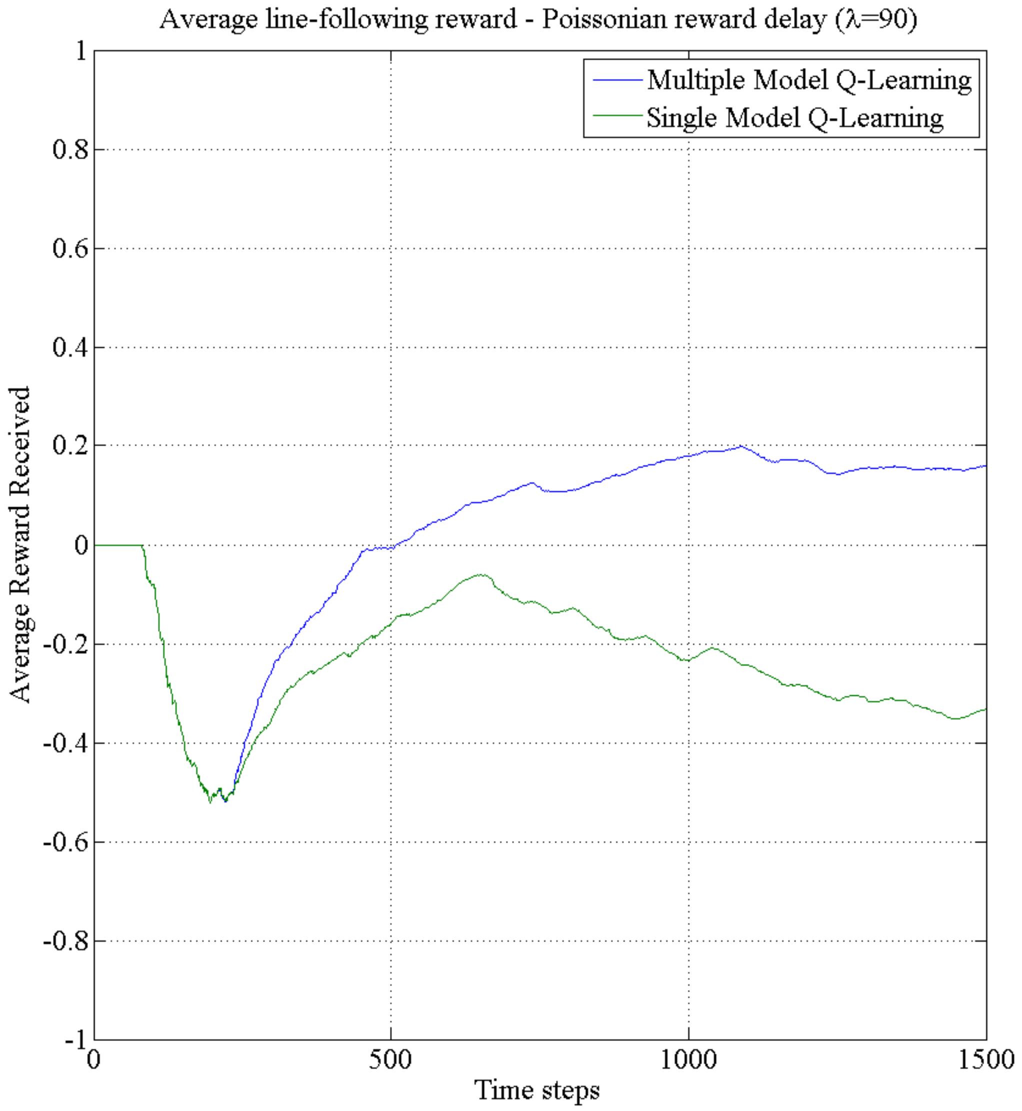


Figure 5.26: Comparison of Average Reward During Experiment 2

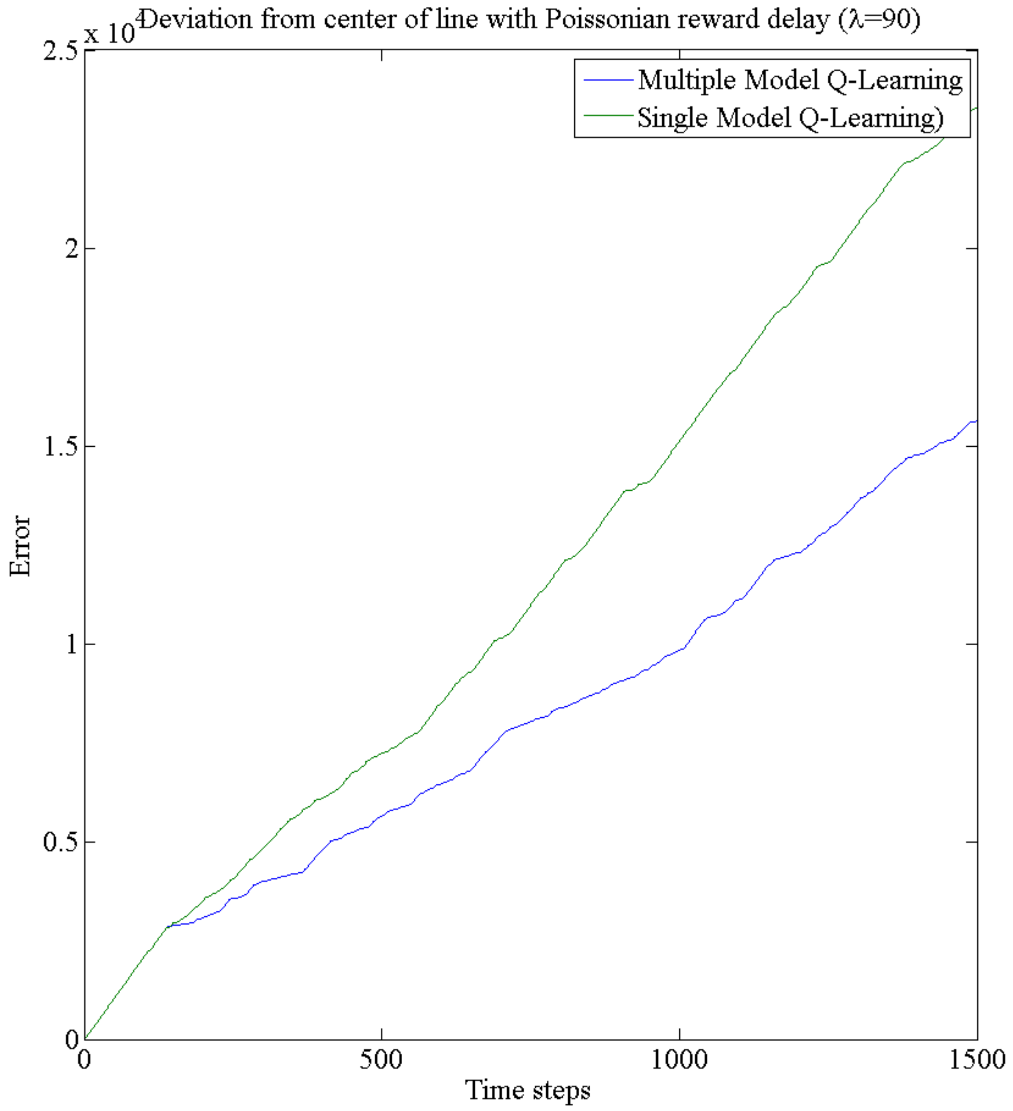


Figure 5.27: Comparison of Accumulated Error During Experiment 2

5.4 Discussion

In this chapter, we have presented the simulation and experimental results of this thesis. Multiple model Q-learning builds upon Q-learning by extending the action space to include an additional time-delay dimension of estimates. This means, as shown in Section 4.1, that the proof of convergence can be extended as well if the definition of value is changed to include an optimization over the new dimension of the time delay in addition to the previous optimization of the agent's real action space. We are now optimizing an estimate, λ^*, and an action, a^*, whereas before we were only optimizing an action, a^*. While Q-learning gives the agent choice over which action to take for a particular state, multiple model Q-learning also gives the agent the choice of a time delay estimate. Thus, the agent has an extra degree of freedom to deal with the stochastic time delayed reinforcement signal present in this problem. Intuitively, the agent can act superstitiously and then test its hypotheses about the time delay using the rewards it receives and the observed state transitions. Over time, time delay estimates which proved to be valuable are used more often and drive the behaviour, increasing the agent's average reward as well.

Beginning with the grid world test bed, this thesis demonstrated how variable reward delays inhibit learning when using Q-learning. We then showed that multiple model Q-learning could still perform, despite the interference of delay, because it is able to learn to adapt to the delay itself.

Next, the grid world results were placed in the Webots simulator, which is a more robust test bed, and the results were consistent with the grid world simulations. Multiple-model Q-learning was able to learn more per unit of experience than could Q-learning in the presence of delays. This is important because real-world training experience can be costly.

Finally, the simulations results were tested in real-world experiments. The e-Puck robots were used on a real-world racetrack printout to test whether the simulated learning could lead to real-world performance. As expected, the e-Puck experiment was more difficult than the simulated Webots environment, but multiple model Q-learning was still able to outperform single-model Q-learning using what it had learned from simulation and combining it with new experience from the real world.

Chapter 6

Conclusion and Recommendations

6.1 Conclusion

As further research is done into decentralized control, such as in networks or swarms of individual mobile robots, applications where it is not safe to assume that reinforcements arrive immediately may become more common. Instead, the reinforcements may be delayed in time due to unpredictable network latency, the need for inexpensive sensors, or the need to calculate reinforcements as a decentralized swarm rather than as an individual.

The work presented in this thesis has argued that multiple-model Q-learning is a suitable method for problems where an agent experiences variable time delays in its reward signal during learning. We showed that there exist applications where the delay interferes to an extent that conventional non-learning controllers, in particular, a simple proportional controller, do not function properly. Since a learning algorithm is required, the novel method allows the learning agent to learn about the time delay itself and assign its rewards accordingly to facilitate more learning than was possible before. The simulation and experimental results in Chapter 5 have shown that a multiple model Q-learning agent facilitates more learning than a Q-learning agent, even when given the same amount of experience and the same amount of time for learning. These techniques permit a designer to imagine a reward function which frees the designer to specify what behaviour the agent ought to have, rather than having to specify *how* to execute the behaviour in detail. This allows for abstraction when designing behaviours, placing the focus on higher-level questions like, "What is it important for the agent to do?" rather than "How do we make an agent do this?"

Inspired by organisms' capability to create superstitions, multiple model Q-learning seeks to establish causal relationships during learning by generating and testing cause-and-effect hypotheses about the time delay itself. The algorithm explores different possible values of the time delay in the same way that Q-learning explores the state-action space. Like a dog seeking treats, the algorithm is able to adapt to the delay over time to increase the probability of assigning its rewards and punishments properly, so as to ensure that it receives more rewards in the future. As a result, multiple model Q-learning learns more efficiently and effectively than single model Q-learning when placed in an environment with variably delayed reinforcements.

6.2 Contributions

Reinforcement learning (RL) problems where reinforcements are stochastically delayed are important because of the growing interest in swarm robotics and decentralized control. In such problems, it may be unclear how much the reinforcement signals will suffer from delay, especially if they are not calculated independently for each swarm member. When the lag between robots or agents, such as in an ad-hoc network, cannot be readily predicted, it becomes necessary for the agents to be able to adapt to the latency itself to maximize learning. As further work is done on decentralized control and machine learning, this type of problem may arise more frequently. In summary, this thesis has contributed to this area of research in the following ways:

1. Introduction of a novel method for learning in environments where stochastic time delays are present in an agent's reward signal based on a formulation of the reinforcement learning problem where variable delays are present in the reward signal, found in section 3.1. The novel algorithm in section 3.2 extends Q-learning and is designed to improve learning despite these delays. The algorithm is able to function even though we have relaxed the condition found in the literature which assumes that reinforcements must arrive in the proper order.

2. Proving the convergence of the novel multiple model Q-learning algorithm in section 4.1, based on a proof of Q-learning in [16] with no condition on reinforcements arriving in the proper order.

3. The following supported these contributions:

   - Demonstration of the problem, evaluation, and comparison of the novel algorithm to Q-learning in a grid world test bed were conducted in section 5.1.

   - Evaluation of multiple model Q-learning in a line-following simulation was carried out to compare multiple-model Q-learning to Q-learning in the Cyberbotics Webots simulator environment. The simulations were done in the Cryberbotics' Webots simulator engine using a simulated e-Puck on a line-following task. The simulations demonstrated that multiple model Q-learning outperforms traditional Q-learning, when given the same amount of experience and time to learn. Performance was measured in terms of the average reward achieved and the accumulated error.

   - Evaluation of the performance of multiple model Q-learning in a real-world test bed, using an e-Puck mobile robot in a line-following task, described in Section 5.3. The experiment showed that multiple model Q-learning outperformed traditional Q-learning, as measured by the average reward obtained and error accumulated during learning, despite having the same amount of time and experience for learning. Learning was faster and of a higher quality. Although multiple model Q-learning has additional computational complexity, the parallel Q-tables can be updated in parallel because they are independent. With additional hardware, the updates could be done in the same amount of time as the updates in traditional Q-learning.

6.3 Recommendations for Future Work

Due to time constraints and the scope of the thesis, several interesting research questions were not addressed in this thesis. Future work might delve into new ways of attributing the rewards into the past to speed convergence. Further research may investigate if function approximators work with the new time delay dimension of the multiple models. The parallel nature of the multiple-models means that the algorithm is well suited to parallelization and future work might explore effective ways of increasing efficiency through parallelization. In addition, it may be possible to apply eligibility traces to the time delay estimates, as has been done in both TD(λ) and Q-Learning.

1. Eligibility Traces: Just as $TD(\lambda)$ and Watkins' Q-learning were able to use eligibility traces to improve the speed of convergence in some cases, this may also be possible for multiple model Q-learning. Eligibility traces could be used in the time delay estimate dimension $\hat{\lambda}$. In this way, delay estimate which had been selected recently in the past could be given a share of the current reward.

2. Parallel computation: Because the parallel Q-tables in multiple model Q-learning are updated independently, the updates can be computed in parallel. With specialized hardware, the algorithm could be optimized to update all parallel Q-tables in the time currently taken to update one Q-table. It is possible that there may be a more efficient order to update the tables, as the implementations herein simply went in numerical order, rather than by order of importance, or value, etc.

3. Reward Kernels: Instead of all parallel tables receiving the current reward equally, a kernel could be used. For example, we might assign full credit for the reward to the model $\hat{\lambda}^*$ which drives behaviour, and then decrease the value of the reward for models which are farther from $\hat{\lambda}^*$. Models which are numerically close to the best estimate will receive proportionally more credit as a result, and appear more valuable and be more likely to be chosen. This might speed convergence.

4. Different Delay Resolutions: Instead of deciding upon a resolution of one time step for the delay estimate $\hat{\lambda}$, we might choose some other value. This would mean that instead of having the set of delay values $\Lambda = (1, 2, 3, 4, ...)$ we might have the values $\Lambda = (3, 6, 9, 12, ...)$, for example. It is possible that a higher resolution would facilitate just as much learning, but require less overall computation because of the lesser number of parallel tables to be updated. This would be increasingly important if the delay were extremely large.

5. Delay Statistics: We have always assumed that the delay statistics are Poissonian. In some applications, it may be important to model the time delay differently. It would be fruitful to study whether multiple model Q-learning could also work with different delay statistics, or if it would have to be modified in some way.

6. Other Applications: There are many other interesting applications which have not been tried in this thesis. Robots in swarms, reinforcement signals involving human reactions and the associated delay times, or shoddy sensors have not been addressed. For example, if the reinforcement signal is derived from some indicator of human emotion, i.e. a reward is a happy human operator and a punishment is an annoyed or angry human operator, then multiple model Q-learning may find a use because human reaction time can be highly variable.

List of References

[1] D. Nguyen-Tuong and J. Peters. “Model learning for robot control: a survey.” *Cognitive Processing* **12**(4), 319–340. ISSN 1612-4782 (2011).

[2] J. Kober and J. Peters. “Reinforcement learning in robotics: A survey.” In M. Wiering and M. Otterlo, editors, “Reinforcement Learning,” volume 12 of *Adaptation, Learning, and Optimization*, pages 579–610. Springer Berlin Heidelberg. ISBN 978-3-642-27644-6 (2012).

[3] R. S. Sutton and A. G. Barto. “Reinforcement learning: Introduction.” (1998).

[4] C. H. Ribeiro. “Embedding a priori knowledge in reinforcement learning.” *Journal of Intelligent and Robotic Systems* **21**(1), 51–71 (1998).

[5] D. Borrajo, L. E. Parker, *et al.* “A reinforcement learning algorithm in cooperative multi-robot domains.” *Journal of Intelligent and Robotic Systems* **43**(2-4), 161–174 (2005).

[6] U. Kartoun, H. Stern, and Y. Edan. “A human-robot collaborative reinforcement learning algorithm.” *Journal of Intelligent & Robotic Systems* **60**(2), 217–239 (2010).

[7] O. Teboul, I. Kokkinos, L. Simon, P. Koutsourakis, and N. Paragios. “Parsing facades with shape grammars and reinforcement learning.” *Pattern Analysis and Machine Intelligence, IEEE Transactions on* **35**(7), 1744–1756. ISSN 0162-8828 (2013).

[8] C. Chen, H.-X. Li, and D. Dong. “Hybrid control for robot navigation - a hierarchical q-learning algorithm.” *Robotics Automation Magazine, IEEE* **15**(2), 37–47. ISSN 1070-9932 (2008).

[9] S. Gonzalez-Valenzuela, S. Vuong, and V. Leung. “A mobile-directory approach to service discovery in wireless ad hoc networks.” *Mobile Computing, IEEE Transactions on* **7**(10), 1242–1256. ISSN 1536-1233 (2008).

[10] I. Arel, C. Liu, T. Urbanik, and A. Kohls. “Reinforcement learning-based multi-agent system for network traffic signal control.” *Intelligent Transport Systems, IET* **4**(2), 128–135. ISSN 1751-956X (2010).

[11] H. Wang, Y. Gao, and X. Chen. “Rl-dot: A reinforcement learning npc team for playing domination games.” *Computational Intelligence and AI in Games, IEEE Transactions on* **2**(1), 17–26. ISSN 1943-068X (2010).

[12] M. Rahimiyan and H. Mashhadi. “An adaptive q -learning algorithm developed for agent-based computational modeling of electricity market.” *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on* **40**(5), 547–556. ISSN 1094-6977 (2010).

[13] V. Chinthalapati, N. Yadati, and R. Karumanchi. “Learning dynamic prices in multiseller electronic retail markets with price sensitive customers, stochastic demands, and inventory replenishments.” *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on* **36**(1), 92–106. ISSN 1094-6977 (2006).

[14] O. K. Sahingoz. “Networking models in flying ad-hoc networks (FANETs): Concepts and challenges.” *Journal of Intelligent & Robotic Systems* **74**(1-2), 513–527 (2014).

[15] K. Katsikopoulos and S. Engelbrecht. “Markov decision processes with delays and asynchronous cost collection.” *Automatic Control, IEEE Transactions on* **48**(4), 568 – 574. ISSN 0018-9286 (2003).

[16] J. N. Tsitsiklis. “Asynchronous stochastic approximation and Q-learning.” *Machine Learning* **16**(3), 185–202 (1994).

[17] L. P. Kaelbling, M. L. Littman, and A. W. Moore. “Reinforcement learning: A survey.” *CoRR* **cs.AI/9605103** (1996).

[18] D. E. Goldberg and J. H. Holland. “Genetic algorithms and machine learning.” *Machine learning* **3**(2), 95–99 (1988).

[19] S. Kirkpatrick, D. G. Jr., and M. P. Vecchi. “Optimization by simmulated annealing.” *science* **220**(4598), 671–680 (1983).

[20] J. Schmidhuber. “A general method for multi-agent reinforcement learning in unrestricted environments.” In “Adaptation, Coevolution and Learning in Multiagent Systems: Papers from the 1996 AAAI Spring Symposium,” pages 84–87 (1996).

[21] B. Argall, B. Browning, and M. Veloso. "Learning robot motion control with demonstration and advice-operators." In "Intelligent Robots and Systems, 2008. IROS 2008. IEEE/RSJ International Conference on," pages 399–404 (2008).

[22] L. G. Valiant. "A theory of the learnable." *Communications of the ACM* **27**(11), 1134–1142 (1984).

[23] C.-N. Fiechter. "Efficient reinforcement learning." In "Proceedings of the seventh annual conference on Computational learning theory," pages 88–97. ACM (1994).

[24] R. S. Sutton. "Learning to predict by the methods of temporal differences." *Machine learning* **3**(1), 9–44 (1988).

[25] P. Dayan. "The convergence of TD (λ) for general λ." *Machine learning* **8**(3-4), 341–362 (1992).

[26] R. 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).

[27] C. J. C. H. Watkins. *Learning from delayed rewards.* Ph.D. thesis, University of Cambridge (1989).

[28] C. J. Watkins and P. Dayan. "Q-learning." *Machine Learning* (1992).

[29] J. Peng and R. J. Williams. "Incremental multi-step q-learning." *Machine Learning* **22**(1-3), 283–290 (1996).

[30] R. S. Sutton. "Integrated architectures for learning, planning, and reacting based on approximating dynamic programming." In "ML," pages 216–224 (1990).

[31] G. Tesauro. *Practical issues in temporal difference learning.* Springer (1992).

[32] D. Chapman and L. P. Kaelbling. "Input generalization in delayed reinforcement learning: An algorithm and performance comparisons." In "IJCAI," volume 91, pages 726–731 (1991).

[33] A. W. Moore. "Variable resolution dynamic programming: Efficiently learning action maps in multivariate real-valued state-spaces." In "Machine Learning: Proceedings of the Eighth International Conference," volume 340. San Francisco, CA, USA Morgan Kaufmann: Los Altos, CA (1991).

[34] V. Gullapalli. "A stochastic reinforcement learning algorithm for learning real-valued functions." *Neural networks* **3**(6), 671–692 (1990).

[35] L. Baird and A. H. Klopf. "Reinforcement learning with high-dimensional, continuous actions." *US Air Force Technical Report WL-TR-93-1147, Wright Laboratory, Wright-Patterson Air Force Base, OH* (1993).

[36] G. Konidaris, S. Osentoski, and P. S. Thomas. "Value function approximation in reinforcement learning using the fourier basis." In "AAAI," (2011).

[37] J. Boyan and A. W. Moore. "Generalization in reinforcement learning: Safely approximating the value function." *Advances in neural information processing systems* pages 369–376 (1995).

[38] J. N. Tsitsiklis and B. Van Roy. "Feature-based methods for large scale dynamic programming." *Machine Learning* **22**(1-3), 59–94 (1996).

[39] S. Mahadevan and J. Connell. "Automatic programming of behavior-based robots using reinforcement learning." *Artificial intelligence* **55**(2), 311–365 (1992).

[40] S. P. Singh. "Reinforcement learning with a hierarchy of abstract models." In "AAAI," volume 92, pages 202–207 (1992).

[41] B. Nemec, M. Zorko, and L. Zlajpah. "Learning of a ball-in-a-cup playing robot." In "Robotics in Alpe-Adria-Danube Region (RAAD), 2010 IEEE 19th International Workshop on," pages 297–301. IEEE (2010).

[42] M. Tokic, W. Ertel, and J. Fessler. "The crawler, a class room demonstrator for reinforcement learning." In "FLAIRS Conference," pages 2471–2482 (2009).

[43] P. Stone, K. Dresner, P. Fidelman, N. K. Jong, N. Kohl, G. Kuhlmann, M. Sridharan, and D. Stronger. *The UT Austin Villa 2004 RoboCup four-legged team: Coming of age.* Computer Science Department, University of Texas at Austin (2004).

[44] T. Yasuda and K. Ohkura. "A reinforcement learning technique with an adaptive action generator for a multi-robot system." In "From Animals to Animats 10," pages 250–259. Springer (2008).

[45] A. Y. Ng, D. Harada, and S. Russell. "Policy invariance under reward transformations: Theory and application to reward shaping." In "ICML," volume 99, pages 278–287 (1999).

[46] S. Russell. “Learning agents for uncertain environments.” In “Proceedings of the eleventh annual conference on Computational learning theory,” pages 101–103. ACM (1998).

[47] T. J. Walsh, A. Nouri, L. Li, and M. L. Littman. “Learning and planning in environments with delayed feedback.” *Autonomous Agents and Multi-Agent Systems* **18**(1), 83–105. ISSN 1387-2532 (2009).

[48] A. Campbell and H. Schwartz. “Multiple model control improvements: hypothesis testing and modified model arrangement.” *Control and Intelligent Systems* **35**(3), 236–243 (2007).

[49] T. Jaakkola, M. I. Jordan, and S. P. Singh. “On the convergence of stochastic iterative dynamic programming algorithms.” *Neural Computation* **6**(6), 1185–1201 (1994).