12-2007

Hardware-Efficient Scalable Reinforcement Learning Systems

Zhenzhen Liu
University of Tennesse - Knoxville

To the Graduate Council:

I am submitting herewith a dissertation written by Zhenzhen Liu entitled "Hardware-Efficient Scalable Reinforcement Learning Systems." I have examined the final electronic copy of this dissertation for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy, with a major in Computer Engineering.

Itamar Elhanany, Major Professor

We have read this dissertation and recommend its acceptance:

Ethan Farquhar, Hairong Qi, J. Wesley Hines

Accepted for the Council:

Carolyn R. Hodges

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

To the Graduate Council:
I am submitting herewith a dissertation written by Zhenzhen Liu entitled "Hardware-Efficient Scalable Reinforcement Learning Systems". I have examined the final electronic copy of this dissertation for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy, with a major in Computer Engineering.

Itamar Elhanany
Major Professor

We have read this dissertation
and recommend its acceptance:

Ethan Farquhar

Hairong Qi

J. Wesley Hines

Accepted for the Council:

Carolyn R. Hodges
Vice Provost and
Dean of the Graduate School

(Original signatures are on file with official student records.)

Hardware-Efficient Scalable Reinforcement Learning Systems

A Dissertation

Presented for the Doctor of Philosophy Degree

Department of Electrical Engineering and Computer Science

The University of Tennessee, Knoxville

Zhenzhen Liu

December 2007

Dedication

This dissertation is dedicated to my parents for their love and support. Thank you.

Acknowledgments

I would like to thank my advisor, Dr. Itamar Elhanany, who brought me into this research area and supported me throughout my studies. I am especially grateful for his perseverance and insightful instruction throughput my time in the program. Thank you.

I would further like to thank Dr. Ethan Farquhar, Dr. Hairong Qi and Dr. J. Wesley Hines, who served on my Ph.D. committee, for their time and input to this dissertation.

Finally and definitely most, I would like to thank my family. I am extremly grateful to my parents, for your love. I dedicate this dissertation to you. Thank you.

Abstract

Reinforcement Learning (RL) is a machine learning discipline in which an agent learns by interacting with its environment. In this paradigm, the agent is required to perceive its state and take actions accordingly. Upon taking each action, a numerical reward is provided by the environment. The goal of the agent is thus to maximize the aggregate rewards it receives over time. Over the past two decades, a large variety of algorithms have been proposed to select actions in order to explore the environment and gradually construct an effective strategy that maximizes the rewards. These RL techniques have been successfully applied to numerous real-world, complex applications including board games and motor control tasks.

Almost all RL algorithms involve the estimation of a value function, which indicates how good it is for the agent to be in a given state, in terms of the total expected reward in the long run. Alternatively, the value function may reflect on the impact of taking a particular action at a given state. The most fundamental approach for constructing such a value function consists of updating a table that contains a value for each state (or each state-action pair). However, this approach is impractical for large scale problems, in which the state and/or action spaces are large. In order to deal with such problems, it is necessary to exploit the generalization capabilities of non-linear function approximators, such as artificial neural networks.

This dissertation focuses on practical methodologies for solving reinforcement learning problems with large state and/or action spaces. In particular, the work addresses scenarios in which an agent does not have full knowledge of its state, but rather receives partial information about its environment via sensory-based observations. In order to address such intricate problems, novel solutions for both tabular and function-approximation based RL frameworks are proposed. A resource-efficient recurrent neural network algorithm is presented, which exploits adaptive step-size techniques to improve learning characteristics. Moreover, a consolidated actor-critic network is introduced, which omits the modeling redundancy found in typical actor-critic systems. Pivotal concerns are the scalability and speed of the learning algorithms, for which we devise architectures that map efficiently to hardware. As a result, a high degree of parallelism can be achieved. Simulation results that correspond to relevant testbench problems clearly demonstrate the solid performance attributes of the proposed solutions.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 The Reinforcement Learning Problem

The reinforcement learning problem [2] is a straightforward framing of the problem of learning from interaction (trial-and-error) to achieve a goal. The learner and decision-maker in this context is called the *agent*. The entity it interacts with, comprising everything outside the agent, is considered its *environment*. These interact continually, the agent selecting actions and the environment responding to those actions and presenting new situations to the agent. The environment also provides rewards, special numerical values that the agent tries to maximize over time.

In most reinforcement learning systems, time is discretized into a sequence of time steps $t = 0, 1, 2, ...$ The interaction between the agent and the environment consists of a sequence of discrete time steps $t = 0, 1, 2, 3....$ At each time step, the agent receives some representation of the environment' state, $s_t \in S$, where S is the set of possible states, and on that basis selects an action $a_t \in A(s_t)$, where $A(s_t)$ is the set of possible actions in state s_t. In the next time step, the environment switches to another state as a sequence of the action and gives the agent an evaluative feedback (reward) r_{t+1}, indicating how good or how bad the immediate effect of the action is. The goal of the agent, generally speaking, is to maximize the total amount of rewards over the long run, which is called return R_t. In this dissertation, the return is defined

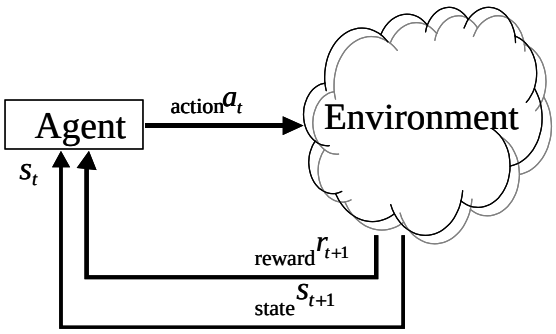

Figure 1-1: Agent-environment interaction diagram

by a discounted sum of the rewards,

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \tag{1.1}$$

where $\gamma (0 \leq \gamma < 1)$ is the discount factor. The agent-environment interaction diagram is depicted in figure 1-1.

At each time step, the agent implements a mapping from states to probabilities of selecting each possible action. This mapping is called the agent's policy and is denoted by π, where $\pi(s_t, a_t)$ is the probability action a_t is selected given that $s = s_t$ under the policy π. Reinforcement learning methods focus on determining how the agent obtains an optimal policy π^*, a policy that maximizes the long term reward, based on its experience. The reinforcement learning framework is a considerable abstraction of the problem of goal-directed learning from experience. It also characterizes the decision making process in a stochastic environment, as will be discussed in more detail throughout this work.

1.1.1 Markov Decision Processes

This thesis focuses on finite-state, discrete-time stochastic dynamic systems. A common assumption in reinforcement learning is that for the environment and task at hand, particularly the state and reward signals, the Markov property holds, that is, the next state and next reward only depend on current state and action [2]. Mathematically expressed, the following is

assumed

$$\begin{aligned} \Pr\{s_{t+1} &= s', r_{t+1} = r | s_t, a_t, r_t, s_{t-1}, a_{t-1}, ..., r_1, s_0, a_0\} \\ &= \Pr\{ s_{t+1} = s', r_{t+1} = r \big| s_t, a_t\}. \end{aligned} \tag{1.2}$$

This assumption can be well justified over a broad range of applications including robotics, automated control, economics and manufacturing, all of which have been shown to exhibit the Markov (memoryless) property.

A reinforcement learning task that satisfies the Markov property is called a (finite) Markov decision process (MDP). MDPs provide a mathematical framework for the study of reinforcement learning algorithms. More formally, an MDP is defined as a (S, A, P, R)-tuple, where S denotes the state space, A contains all the possible actions at each state, P is a probability transition function $S \times A \times S \to [0, 1]$ and R is the reward function $S \times A \times S \to R$. Moreover, the policy π is a mapping from the state set to the action set: $\pi : S \to A$. The probability transition function P is given by

$$P^a_{ss'} = \Pr\{s_{t+1} = s' | s_t = s, a_t = a\} \tag{1.3}$$

for any $s, s' \in S, a \in A$. Likewise, the reward function is

$$R^a_{ss'} = E\left\{r_{t+1} | s_t = s, a_t = a, s_{t+1} = s'\right\} \tag{1.4}$$

where $E\{*\}$ denotes the expected value of the next reward. The functions P and R entirely characterize the dynamics of a finite MDP. They are also assumed constant if the environment is stationary.

1.1.2 Partially Observable Markov Decision Process (POMDP)

Fully observable MDPs assume that the state information is accessible to the decision maker in the decision making process, in other words, the decision maker knows exactly which state it is in when interacting with its environment. However, there is a broad range of real-world problems in which an agent interacts with its environment without being provided with an explicit state

representation, or the state space is not directly or fully observable. A typical example would be the path planning problems for mobile robots, sometimes called the "Kidnapped Robot Problem", by imagining a robot was moved to an unknown location in a known environment and now must figure out where it is and find its way home. In this case, the agent only has observations of its position in its vicinity in stead of where it is in the whole map [3].

An exact solution to the POMDP will generate the series of actions that is most likely to get it home with the least cost. An POMDP is defined as a (O, A, P, R)-tuple, where O denotes the observation space, and A, P, R represent action space, transition probabilities, and rewards, respectively, all corresponding to an MDP. The policy in a POMDP maps the observation set to the action set $\pi : O \rightarrow A$.Deriving the optimal policy is achieved through state inference, in other words, by considering the distribution of states for a given observation. Let the observation at time step t be o_t, then the reward for action a_t at o_t is expressed by the average reward for all possible states of o_t[3], such that

$$r_{t+1}(o_t, a_t) = \sum_s \Pr\{s|o_t\} r_{t+1}(s, a_t). \tag{1.5}$$

A broad range of POMDPs are related to state inference from a series of past observations and actions. A typical example of this is the robot navigation, where the agent may receive identical observations for several different positions (or states). In these cases, the agent must recall recent steps in order to infer its precise position. Therefore, the agent should maintain internal representation of the past history during its execution of a sequential task. This dissertation serves as an attempt to devise efficient solutions to these type of problems.

1.1.3 Value Functions

Almost all reinforcement learning methods are based on estimating value functions – functions of state (or of state-action pairs) that evaluate how good it is for the agent to be in a given state (or how good it is to perform a given action in a given state) [2]. The notion of "how good" here is defined in terms of future rewards that can be expected, or, to be precise, in terms of expected return. Recall that a policy maps the state space to the action space through the probability $\pi(s, a)$, denoting the probability of choosing action a in state s. To evaluate a

policy, we compute the value of the state s under the policy π,denoted by $V^\pi(s)$, which is the expected return when starting in s and following π thereafter. For an MDP, we have

$$V^\pi(s) = E_\pi \{R_t | s_t = s\} = E_\pi \left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \middle| s_t = s \right\}. \tag{1.6}$$

Similarly, the value of taking action a in state s under a policy π, denoted by $Q^\pi(s, a)$, is defined as the expected return starting from s, taking the action a, and following policy π thereafter:

$$Q^\pi(s, a) = E_\pi \{R_t | s_t = s, a_t = a\} = E_\pi \left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \middle| s_t = s, a_t = a \right\}. \tag{1.7}$$

V^π and Q^π are called the state-value function for policy π and action-value function for policy π, respectively. The solution of an MDP is an optimal policy π^* that maximizes the action-value functions,

$$\pi^*(s) = \arg \max_{a \in A(s)} (Q^*(s, a)). \tag{1.8}$$

1.2 Reinforcement Learning Methods

Reinforcement learning categorize a range of tractable approximation algorithms for solving MDPs. Reinforcement learning algorithms attempt to find a policy that maps states of the world to the actions the agent ought to take in those states. The environment is formulated as a finite-state MDP, and reinforcement learning algorithms of this context are highly related to dynamic programming techniques. State transition probabilities and reward probabilities in MDPs are typically stochastic but stationary over the course of the problems. However, in many practical scenarios, the transition probability $P_{ss'}(a)$ and the reward function $R(s, \pi(s))$ are unknown, which makes it hard to evaluate the policy π.

1.2.1 Dynamic Programming

The term dynamic programming (DP) [4] refers to a collection of algorithms that can be used to compute optimal policies given a perfect model of the environment as a MDP. Classical DP algorithms are of limited utility in reinforcement learning both because of their assumption of a perfect model and because of their great computational expense, but they are still important

theoretically. In fact, the reinforcement learning algorithms listed in this thesis can be viewed as attempt to achieve much the same effect as DP, only with less computation and without assuming a perfect model of the environment. The key idea behind DP, and reinforcement learning in general, is the use of value functions to organize and structure the search for good policies. From the definition of $V^\pi(s)$, we have:

$$\begin{aligned} V^\pi(s) &= E_\pi \{R_t | s_t = s\} \\ &= E_\pi \left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \middle| s_t = s \right\} \\ &= E_\pi \left\{ r_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k r_{t+k+2} \middle| s_t = s \right\} \\ &= E_\pi \left\{ r_{t+1} + \gamma V^\pi(s_{t+1}) \middle| s_t = s \right\} \end{aligned} \tag{1.9}$$

DP has two phases: *policy evaluation* and *policy improvement*. The state value function V_0 is initialized arbitrarily, and each successive approximation is obtained by using the Bellman equation for V^π above as an update rule:

$$\begin{aligned} V^\pi_{k+1}(s) &= E_\pi \left\{ r_{t+1} + \gamma V^\pi(s_{t+1}) \middle| s_t = s \right\} \\ &= \sum_a \pi(s,a) \sum_{s'} P^a_{ss'} \left[R^a_{ss'} + \gamma V_k(s') \right] \end{aligned} \tag{1.10}$$

for all $s \in S$. The fixed point of this equation is V^π: the actual state value function for policy π. Also, it has been shown that, generally, as $k \to \infty$ the serial $\{V_k\}$ converges to V^π. This algorithm is called iterative policy evaluation. The next phase is policy improvement, in which we determine the best action at each state based on V^π. Now that we have the value function V^π for an arbitrary deterministic policy π, we want to know if it is better to switch to another policy, in other words, to choose an action $a \neq \pi(s)$ in state s. To do this, we consider selecting a in s and thereafter following the existing policy π,the action value function becomes:

$$\begin{aligned} Q^\pi(s,a) &= E_\pi \left\{ r_{t+1} + \gamma Q^\pi(s_{t+1}, a_{t+1}) \middle| s_t = s, a_t = a \right\} \\ &= \sum_{s'} P^a_{ss'} \left[R^a_{ss'} + \gamma V_k(s') \right] \end{aligned} \tag{1.11}$$

To improve the policy, at each state s, we compute the action value for each possible action

The Dynamic Programming Algorithm

1. Initialization
   $V(s) \in R$ and $\pi(s) \in A(s)$ arbitrarily for all $s \in S$
2. Policy Evaluation
   Repeat
     $\Delta \leftarrow 0$
     For each $s \in S$:
       $v \in V(s)$
       $V(s) \leftarrow \sum_{s'} P_{ss'}^{\pi(s)} \left[R_{ss'}^{\pi(s)} + \gamma V(s') \right]$
       $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
   until $\Delta < \theta$ (a small positive number)
3. Policy Improvement
   policy-stable $\leftarrow$ true
   For each $s \in S$
       $b \leftarrow \pi(s)$
       $\pi(s) \leftarrow \arg\max_a \sum_{s'} P_{ss'}^{\pi(s)} \left[R_{ss'}^{\pi(s)} + \gamma V(s') \right]$
       if $b \neq \pi(s)$, then *policy-stable* $\leftarrow$ false
   if *policy-stable*, then stops; else go to 2

Table 1.1: Dynamic programming algorithm.

and choose the one with the best $Q^\pi(s, a)$. In other words, the new *greedy* policy is selected:

$$\begin{aligned} \pi'(s) &= \arg\max_a Q^\pi(s, a) \\ &= \arg\max_a \sum_{s'} P_{ss'}^a \left[R_{ss'}^a + \gamma V_k(s') \right] \end{aligned} \tag{1.12}$$

DP consists of a serial of interweaved policy evaluation and policy improvement: once a policy π, has been improved using V^π to yield a better policy π', we then compute $V^{\pi'}$ and improve it again to yield an even better π''. Each policy is guaranteed to be a strict improvement over the previous one (unless it is already optimal). A complete algorithm is given in table 1.1 [2].

The disadvantages of DP is that it assumes full knowledge of system dynamics including the reward function R and transition probability function P, and also the computational complexity involved is overwhelming.

1.2.2 Temporal-Difference Learning

Temporal Difference (TD) learning [2] has been mostly used for solving the reinforcement learning problems. TD methods require only experience–sample sequences of states, actions and reward from on-line or simulated interaction with an environment. TD is related to dynamic programming techniques since it approximates its current value estimate based on previously learned estimates (a process known as bootstrapping). Another way of looking at TD is of learning from guess to guess. As a prediction method, TD learning takes into account the fact that subsequent predictions are often correlated in some sense. In standard supervised predictive learning, one only learns from actually observed values, a prediction is made, and when the observation is available, the prediction is adjusted to better match the observation. The core idea behind TD learning is that we adjust predictions to match other, more accurate predictions, about the feature.

TD learning bases its update process in part on an existing estimate and can thus be used to estimate value functions. This can be expressed formally as

$$V(s_t) \leftarrow V(s_t) + \alpha[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)] \tag{1.13}$$

where r_{t+1} is the observed reward at time $t+1$, $\alpha(0 < \alpha < 1)$ is the learning rate parameter, and $[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)]$ is called the *temporal difference*. The TD method is called a "bootstrapping" method, because the value is updated partly using an existing estimate and not a final reward.

These methods use sample backups, which are different from backups of DP methods in that they are based on a single sample successor rather than on a complete distribution of all possible successors. In the context of a control problem, TD methods can be used to evaluate and to predict the action-value function under a given current policy. For the action value function, we have

$$Q^{\pi}(s,a) = E_{\pi}\left\{ r_{t+1} + \gamma Q^{\pi}(s_{t+1}, a_{t+1}) \middle| \, s_t = s, a_t = a \right\}. \tag{1.14}$$

Substituting state-action variables for state variables, the updating rule becomes

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]. \tag{1.15}$$

This algorithm is called Sarsa, for which the update is done after every transition from a non-terminal state s_t. If s_{t+1} is terminal, then $Q(s_{t+1}, a_{t+1})$ is defined as zero.

$Q-$Learning

Q-Learning [5] is one of the most effective and popular algorithms for learning from delayed reinforcement to determine an optimal policy, in the absence of the transition probability and reward function. The update rule for one-step $Q-$learning is defined by

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)]. \tag{1.16}$$

The difference between $Q-$learning and Sarsa is that $Q-$learning is an off-policy method, in which the learned action-value function Q, directly approximates Q^*, the optimal action-value function, independent of the policy being followed. The policy still has an effect in that it determines which action to take in each state. It has been proven that $Q-$learning converges faster than Sarsa [2].

1.2.3 Generalization and Function Approximation

Tabular form reinforcement learning cannot handle applications with large scale state and action spaces, in particular when the states and actions are continuous. When the state and/or action spaces are large, estimates of the value function cannot be represented in a table with one entry for each state or each state-action pair. An exponential growth in the size of the state or action sets is observed as their dimensions increase. This is often referred to as the curse of dimensionality, a well-known phenomenon in many fields, including pattern-recognition and machine learning. The problem is not just the memory needed for large tables, but the time and data needed to accurately fill them. In other words, the key issue is that of generalization. How can experience with a limited subset of the state space be usefully generalized to produce a good approximation over a much larger subset?

This is, indeed, a complicated and intricate problem. In many tasks, to which we would like to apply reinforcement learning, most states encountered will never have been experienced exactly before. This will almost always be the case when the state or action spaces include continuous variables or large number of sensors, such as a visual image. The only way to learn anything at all on these tasks is to generalize from previously experienced states to ones that have never been seen.

Fortunately, generalization from examples has already been extensively studied, and we do not need to invent totally new methods for use in reinforcement learning. To a large extent we need only combine reinforcement learning methods with existing generalization methods. The kind of generalization we require is often called function approximation because it takes examples from a desired function (e.g., a value function) and attempts to generalize from them to construct an approximation of the entire function. Function approximation is an instance of supervised learning, a primary topic studied in machine learning, artificial neural networks, pattern recognition, and statistical curve fitting.

1.3 Motivation

In view of the above, this dissertation aims to focus on solving MDPs which are characterized by large state space and action space. Moreover, in an effort to address realistic machine learning scenarios, this work will also study ways in which POMDPs can be solved. In order to achieve these goals, we will consider both tabular as well as function-approximation based reinforcement learning frameworks. A pivotal theme of this work is the hardware consideration perspective, which sets as a goal to derive architectures which map to hardware so as to yield highly scalable reinforcement learning solutions.

1.4 Dissertation Outline

In the following four chapters we will describe previous works targeting solutions for reinforcement learning problems with large state and action space and then introduce our novel techniques for addressing such large scale problems. Chapter 2 provides an overview of Neuro-Dynamic programming - a framework for solving POMDPs using recurrent neural networks as

function approximators. We will outline the core limitations of such methods, in particular in the context of resource requirements.

In chapter 3, we describe a pipelined Q-learning architecture as an attempt to scale Q-learning with finite state space and large action spaces, which also induces delays. To complement the design, convergence proofs are provided for the proposed scheme. Chapter 4 introduces a novel variation of real-time recurrent learning (RTRL), a learning algorithm for recurrent neural networks, called Truncated-RTRL. The latter aims to reduce the computational complexity and storage requirement of RTRL and to increase the learning rate, based on stochastic gradient descent methodologies. Further, for hardware realization purposes, we localize TRTRL by clustering neurons as a trade-off between connectivity and performance. Chapter 5 introduces a consolidated actor-critic model (CACM) for a simplified model-free temporal difference learning. Chapter 6 provides a summary of the contributions made.

Chapter 2

Literature Review

2.1 Recurrent Neural Networks

Recurrent neural networks (RNNs), first described in [6], are fundamentally different from feedforward architectures in the sense that they operate not just on an input space but also on an internal state space. Figure 2-1 illustrates the block diagram of the state-space generic recurrent network. RNNs are widely acknowledged as an effective tool that can be used by a wide range of applications that store and process temporal sequences. The ability of RNNs to capture complex, nonlinear system dynamics has served as a driving motivation for their study. RNNs have the potential to be effectively used in a wide range of modeling, system identification and control applications, where other techniques may fall short. Consequently, a variety of learning algorithms have been proposed, the majority of which rely on the calculation of error gradients with respect to the network weights. What distinguishes recurrent neural networks from static, or feedforward networks, is the fact that the gradients are time-dependent or dynamic. This implies that the current error gradient does not only depend on the current input, output and targets, but rather on its possibly infinite past. How to effectively train RNNs remains a challenge and an active research topic.

RNNs are neural networks which utilize recurrent links in order to provide dynamic memory. The recurrent connections allow the network's hidden units to see its own previous output, so that future behavior can be shaped by previous responses. There are many types of RNNs but they all have two common features. All RNNs make use of some part of the static multilayer

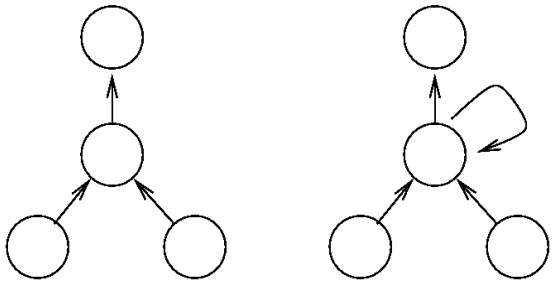

Figure 2-1: A simple feedforward network and a recurrent network with an input layer, one hidden layer containing one processing element, and an output layer.

perceptron feedforward network and exploit the nonlinear mapping capability of the multilayer feedforward model. The basic distinction between feedforward static networks and RNNs is shown in Figure 2-1. Recurrent neural networks have a feedforward connection for all neurons which allow the network to show dynamic behavior. A network with a fully connected hidden layer, between the input layer and the output layer is depicted in figure 2-2

Practical constraints often guide the selection of one RNN learning algorithm over another. The learning problem consists of adjusting the parameters (or *weights*) of the network, so that the trajectories have certain specified properties. A common learning algorithm is known as backpropagation. In backpropagation, the weights of the neural network can be adjusted so as to produce an output on the appropriate unit when a particular pattern at the input is observed. The algorithm works by running the training instance through the neural network, and calculating the error between the desired (target) and actual outputs. These differences are then "propagated back" from the output layer to the hidden and input layers in the form of modifications to the weights of each of the component neurons. We next review the primary RNN architectures, and associated learning rules.

2.1.1 Elman Neural Networks

The simple recurrent network (SRN) described in [7] and depicted in Figure 2-3, has an architecture similar to that of Figure 2-2, with the exception that the output layer may be nonlinear and the bank of unit delays at the output is omitted. The Elman approach calls the bank of

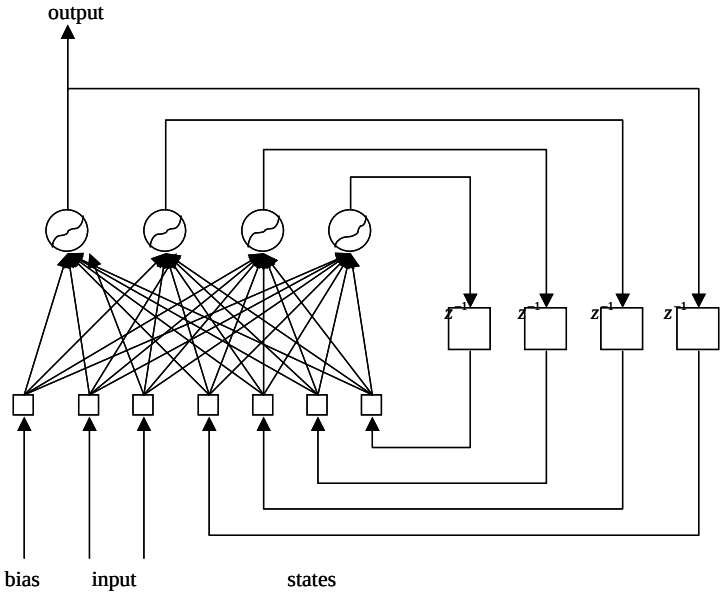

Figure 2-2: A full connected recurrent neural network

unit delays context units, which are also "hidden" because they interact solely with other nodes in the network and not with the outside world. Network processing consists of the following sequence of events. At time t, the input units receive the first input in the sequence. The hidden units feed forward to the output units and at the same time, feed back to the context units. The context units then store the output of the hidden units for one time step, and then feed them back to the input layer. Based on this description, there is only a feedforward cycle, but a learning phase using backpropagation [8] may be used.

By utilizing hidden units and a learning algorithm, the hidden units develop internal representations for the input patterns. These neurons continue to recycle information through the network over multiple time steps, and thereby discover abstract representations of time. Therefore, we say that the context units provide the network with dynamic memory so as to encode the information contained in the input pattern and remember the previous internal state. In the following section, we will briefly review the primary backpropagation-based algorithms used to update the weights in RNNs.

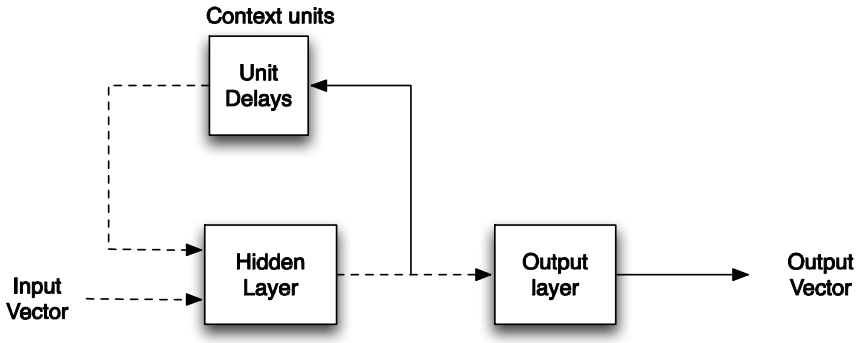

Figure 2-3: The Elman simple recurrent network where activations are copied from the hidden layer to the context layer and then fed back into the hidden layer after a one time step delay. The dotted lines represent trainable connections.

2.1.2 Backpropagation Through Time

The backpropagation through time (BPTT) algorithm can be viewed as an extension to the classical Elman network described in section 2.1.1 and is a generalization of backpropagation for static networks. Various batch-training forms of the algorithm have been derived by [9]. Other versions were derived and discussed in [8].

Let $\mathcal{N}$ denote a recurrent neural network required to learn a temporal task, starting at time t_0 and ending at time t. Next, let us denote $\mathcal{N}^*$ as the feedforward network that results from unrolling the temporal operation of the recurrent network $\mathcal{N}$, where $\mathcal{N}^*$ has a layer for each time step in the time interval $[t_0, t]$ and n units in each layer. For each neuron in network $\mathcal{N}$, there is a copy of a layer in $\mathcal{N}^*$. Every connection from unit i to unit j in $\mathcal{N}$ has a corresponding connecting unit j in layer l to unit i in layer $l+1$, for each time step $l \in [t_0, t]$.

During the first phase, a copy of the entire RNN is added to the top of a growing feedforward network on each update cycle, which updates the internal states of the network. Thus, if the network is to process a signal that is t time steps long, then copies of the network are created and the feedback connections are modified such that there are feedforward connections from one network to the subsequent network. Second, backpropagation is used to update the weights with respect to the performance error. In a subsequent phase, the network is trained using backpropagation to update the weights with respect to the performance error. It becomes one large feedforward network with the updated weights being treated as shared weights.

The key advantage of BPTT is that the training algorithm, backpropagation, is identical to those that are used for feedforward networks and therefore it can be applied to a wide variety of problems. However, the epochwise BPTT algorithm [10] has several fundamental drawbacks: first, it is not a real-time algorithm in the sense that batch data must be applied and second, the algorithm has extensive memory requirements that are dictated by the need to store growing amounts of state information. The procedure works well for relatively simple recurrent networks consisting of a few neurons as it has a computational complexity of $O(N^2)$, however the memory requirements of the underlying formulas become too large when the procedure is applied to more general architectures that are typical of those encountered in practice. Other, continuous time approaches to training recurrent networks to handle time-varying input or output have been investigated by [11]. Unfortunately, these approaches use a restrictive architecture that is not suitable for more complex problems.

2.1.3 Real Time Recurrent Learning

The real-time recurrent learning (RTRL) algorithm [12] is possibly the most popular weight updating scheme for RNNs, and will be used as basis for some of the core contributions of this dissertation. Let us assume that a network consists of a set of N fully connected neurons and a set of M inputs. Let $w_{ij}(t)$ denote the weight (i.e. the synaptic strength) associated with the link originating from neuron j towards neuron i at time t. The net input to neuron k, $s_k(t)$, is defined as the weighted sum of all activations in the network, $z_l(t)$. Based on standard RTRL terminology, we define the activation function of node k at time $t+1$ to be

$$y_k(t+1) = f_k\left(s_k\left(t\right)\right), \tag{2.1}$$

where

$$s_k(t) = \sum_{l \in N \cup M} w_{kl} z_l(t), \tag{2.2}$$

$$z_k(t) = \begin{cases} x_k(t) & \text{if } k \in M \\ y_k(t) & \text{if } k \in N \end{cases} \tag{2.3}$$

and the non-linear activation function, $f(\cdot)$, maps $s_k(t)$ to the range [0,1]. The overall network error at time t is defined by

$$J(t) = \frac{1}{2} \sum_{k \in \text{outputs}} [d_k(t) - y_k(t)]^2 = \frac{1}{2} \sum_{k \in \text{outputs}} [e_k(t)]^2 \tag{2.4}$$

where $d_k(t)$ denotes the desired target value for output k at time t. Correspondingly, the error is minimized along a negative multiple of the performance measure gradient. The online calculation of the gradients is achieved by exploiting the following relationship:

$$-\frac{\partial J(t)}{\partial w_{ij}(t)} = \sum_{k \in \text{outputs}} e_k(t) \frac{\partial y_k(t)}{\partial w_{ij}}. \tag{2.5}$$

By identifying the partial derivatives of the activation functions with respect to the weights as sensitivity elements, and denoting the notation by

$$p_{ij}^k(t) = \frac{\partial y_k(t)}{\partial w_{ij}}, \tag{2.6}$$

we obtain the following recursive equation:

$$p_{ij}^k(t+1) = f_k'\left(s_k\left(t\right)\right) \left[\sum_{l \in N} w_{kl} p_{ij}^l(t) + \delta_{ik} z_j(t) \right], \tag{2.7}$$

where $p_{ij}^k(0) = 0$ and δ_{ik} is the Kronecker delta. Equations (2.7) and (2.5) allow one to obtain the performance gradient at any given time. Finally, the updating rule is given by

$$w_{ij}(t+1) = w_{ij}(t) + \alpha \sum_{k \in \text{outputs}} e_k(t) p_{ij}^k(t),$$

where α is the learning rate parameter.

As can be seen from these equations, each neuron is required to perform $O(N^3)$ multiplications yielding an overall complexity of $O(N^4)$. Moreover, the storage requirements are dominated by the weights $O(N^2)$ and, more importantly, the sensitivity matrices, $p_{ij}^k(t)$, which are $O(N^3)$. Due to the distributed nature of the network, the calculation can be reduced significantly by having each neuron compute its sensitivities in parallel. If performed in hardware,

these computation processes can be accelerated by exploiting pipelining and module replication. However, unlike the computational requirements, the storage requirements cannot be reduced as they constitute a crucial component in the weight update procedure.

Several schemes that have been presented in the literature aim to reduce the storage complexity associated with RTRL. A unifying theme of these methods comprises of subgrouping the neurons into multiple, non-overlapping subnetworks. Although the computational gain is significant, the storage requirements remain high, in particular when a small set of subgroups is employed.

2.2 Neuro-Dynamic Programming

Neuro-dynamic programming (NDP), also called Approximate Dynamic Programming, is a new class of dynamic programming methods for control and sequential decision making under uncertainty. These methods have the potential of solving the problems that for a long time were thought to be intractable due to either a large state or action space, or the lack of an accurate model. NDP methods are suboptimal methods that center around the approximate evaluation of optimal cost function through the use of neural network and/or simulation. They aim at developing a methodological foundation for combining dynamic and compact representation in order to derive an optimal or suboptimal solution for MDPs.

2.2.1 Approximation Architecture: Neural Networks

Many researchers have proposed the use of neural networks as function approximating architecture in the context of NDP. Neural networks here are not restricted to the classical multilayer perceptron structure with sigmoidal nonlinearities, but rather any type of universal perceptron structure of nonlinear mappings could be used in this context. Recurrent neural networks are convenient in that they are able to model dynamic systems with memory. They are used to approximately evaluate the value function in NDP. The states, or state-action pairs, are mapped into a feature vector, which is then fed into a neural network as inputs to produce a score of the state or state-action. The generalization ability of neural networks allows classification of high-dimensional state-action inputs into value functions.

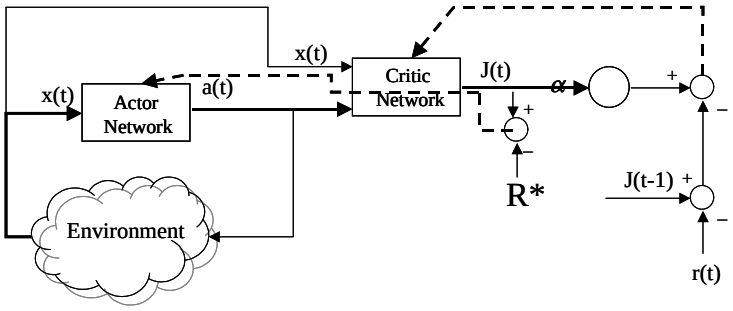

Figure 2-4: Direct neural dynamic programming diagram. The solid lines denote system flow, while the dashed lines represent error backpropagation paths for critic and actor networks.

2.2.2 Direct NDP: The Actor-critic Architecture

By using neural networks as value function approximators in the RL framework, an actor-critic learning architecture has been proposed in [13] to implement direct NDP. Fig 2-4 shows the schematic diagram for implementation of direct NDP. The objective of this on-line learning control scheme is to optimize a desired performance measure by learning to choose appropriate control actions through interaction with the environment. The direct NDP structure includes two networks, actor and critic as building blocks. The critic uses an approximation architecture to learn a value function, which is then used to update the actor's policy parameters in the direction of performance improvement. Both of the actor and critic networks are initialized with random parameters (weights). Once a system state $s(t)$ is observed, the state information $x(t)$ is fed into the actor network to generate an action $a(t)$. The action value function, $J(t)$, will then be computed based on the parameters of the critic network. Finally, the Bellman error $([r(t) + \alpha J(t)] - J(t-1))^2$ is utilized in order to tune the weights of the critic network. Adaptation of the actor network is done by back-propagating the error between the desired ultimate performance objective R^* and the approximate function J from the critic network.

The learning algorithm is implemented by the pseudocode below. $\mathbf{nn}_a$ and $\mathbf{nn}_c$ denote the actor network and critic network, respectively, while e is the error, w the weights/parameters and l the learning rate. Updating of the weights in the actor network is achieved by back-propagating the error via the critic network and the action signal.

NeuroDynamic Programming Algorithm
Initialize w_a, w_c arbitrarily
Repeat (for each time step t)
Observe state $s(t)$, extract feature vector $x(t)$
Choose action $a(t) \longleftarrow \mathbf{nn}_a(x(t))$
Take action $a(t)$, observe $r(t)$, $s(t+1)$
Update w_c
$e_c(t) \longleftarrow \alpha J(t) - [J(t-1) - r(t)]$
$E_c(t) \leftarrow \frac{1}{2} e_c^2(t)$
$\Delta w_c(t) \leftarrow l_c(t) \left[-\frac{\partial E_c(t)}{\partial w_c(t)} \right]$
$w_c(t+1) \leftarrow w_c(t) + \Delta w_c(t)$
Update w_a
$e_a(t) \leftarrow J(t) - R^*$
$E_a(t) \leftarrow \frac{1}{2} e_a^2(t)$
$\Delta w_a(t) \leftarrow l_c(t) \left[-\frac{\partial E_c(t)}{\partial a(t)} \frac{\partial a(t)}{\partial w_a(t)} \right]$
$w_a(t+1) \leftarrow w_a(t) + \Delta w_a(t)$
$s(t) \leftarrow s(t+1)$, $J(t-1) \leftarrow J(t)$
Until the performance objective is met

Table 2.1: A Generic NeuroDynamic Programming algorithm.

2.3 Solving POMDPs

Many problems of interest can be formulated as POMDPs, yet the lack of efficient algorithms results in the limited use of POMDPs in practice. In MDPs the agent's observation is equivalent to the environment's state. Therefore, the solution for MDPs is simply a mapping between observed states to actions. However, in a POMDP, such a memoryless or perception-based policy will not suffice, and the agent must learn an internal state-based policy. In [14] [8] recurrent neural networks are considered for solving POMDPs, by inferring state information as means of approximating the value function. Also, [15] investigates an actor-critic architecture, where both actor and critic are fully recurrent neural networks, both trained with RTRL.

In [16] Elman recurrent neural networks are trained with standard backpropagation, but only for obtaining a direct reward. Finally, [17] also use Elman networks, which approximate the Q-learning's value function and are trained using BPTT.

Generally speaking, the majority of the work done on solving POMDPs with RNNs has been limited in capacity and scale. This is primarily due to the inherent scalability limitations of existing RNN technologies. Moreover, convergence to optimal policy is not guaranteed when

using RNNs for value approximation. This dissertation aims to address these key issues by proposing an RNN framework for solving complex POMDPs in a manner that scales and delivers adequate performance characteristics.

Chapter 3

Large-scale Tabular-form Reinforcement Learning Architectures

3.1 Q-Learning Hardware Architecture

In many applications the action set is rather large. This is particularly true for robotics, where high-dimensional output signals may exist. For example, a robot may have an action vector of 12 elements with each having 8 possible values, resulting in an action set size of 8^{12}. To address such cases, this chapter presents a framework for hardware-oriented tabular form Q-learning. In particular, the proposed architecture targets applications with a finite state space and a high-dimensional action space. Applying $Q-$Learning in this context would introduce a significant delay in determining the action for each state according to the policy (either softmax or ε greedy), which we shall refer to as *action delay*. The latter originates from the operator

$$a^* = \arg\max_a Q(s, a), \tag{3.1}$$

which is common to most Q-learning variants. Thus, for real-time $Q-$Learning applications, the bottleneck is the max action selection. Further, the bootstrapping process also suffers from

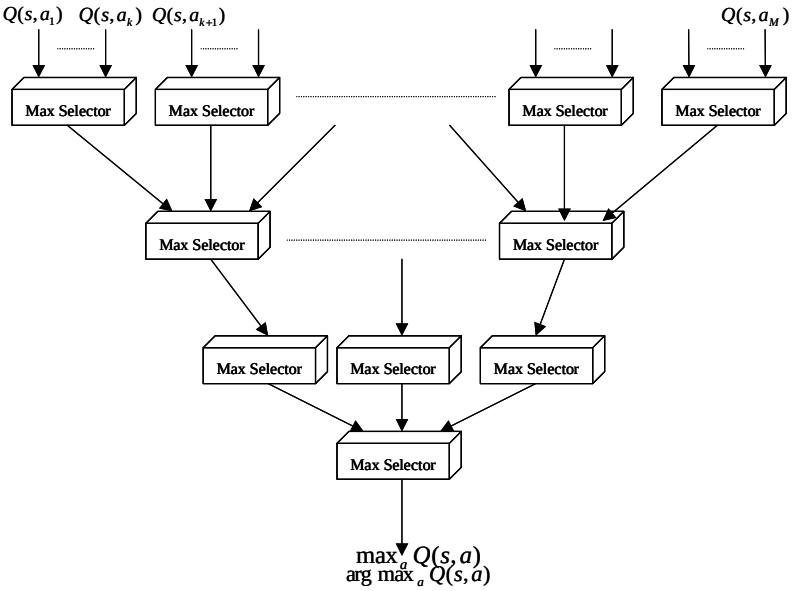

Figure 3-1: Pipelined structure for maximal action selection

delay in calculating the maximum of a next state action value, as reflected by

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)]. \tag{3.2}$$

Figure 3-1 depicts the pipelined diagram proposed for Q-Learning with large action sets. Suppose the size of the action set for each state is M, and the maximum of k values can be found by each block, by exploiting the pipelined architecture the delay in action selection is $O(\log_k M)$. It becomes apparent that there is a delay between the instant an observation is received and the time its corresponding action is selected by the agent. In the following section, we provide proof that $Q-$Learning with such delays also converges to the optimal policy.

3.2 Convergence of Q-Learning with Delays

The basic MDP formulation introduced in section 1.1.1 is inadequate for some control problems for the following reasons. First, observation delays may exist, for which information pertaining to the state of a system arrive with delay rather than being available instantaneously. Second,

action delays may exist whereby actions take effect at a later time rather than immediately after being issued. Finally, the cost induced by an action may be collected after a number of stages. Hence, MDPs with delays can be used to model dynamic environments in those scenarios [18]. Such models have been applied to a number of control problems, such as communication network design [19], transportation information network design [20] and decentralized control problems [21] [22]. It has been shown that MDPs with delays can be reduced to equivalent MDPs without delay [19] [18] for which the optimal policies are attained through the equivalent MDPs using dynamic programming techniques [23]. Our motivation in this work is to solve MDPs with delays when the reward function g and transition probabilities P_A are unknown. As such we focus our attention on Q-learning[24] - the prevailing off-policy algorithm for addressing such scenarios.

Q-Learning was proposed as a method for solving MDPs with unknown costs and transition probabilities. It utilizes simulation or real-time experience to iteratively approximate the state-action value function. $Q-$learning has been widely applied in market control[25], fuzzy logic control[26], robot soccer systems[27], to name a few. The convergence of Q-Learning to optimal policies has been proven in [28], however, the proof relies on the fundamental assumption that the underlying MDPs have no delays. In particular, the following two assumptions are made: (1) the current system state is available to the decision making agent without any delay, and (2) the actions issued by the agent take effect instantaneously while the rewards are collected at the succeeding stages. However, in MDPs with delays, both assumptions are violated.

We consider MDPs with finite state and action spaces having bounded delays. We then formulate the Q-Learning algorithm for MDPs with delays and identify its optimal values so as to obtain respective convergence properties. The rest of this chapter is structured as follows. In Section 3.3 we present analysis pertaining to the case of constant delays for observations and actions. Section 3.4 extends these results to address the case of random delays.

3.3 Constant Delays

An underlying embedded MDP with constant observation delay, o, constant action delay, ac, and constant cost delay, c, can be denoted as a seven-tuple $< S, A, P_A, g, o, ac, c >$, referred

to as a deterministic delayed MDP (DDMDP) [19]. We utilize this definition in formulating Q-Learning for DDMDPs by augmenting the state space and identifying its optimal values. Based on the latter, a convergence proof is derived.

3.3.1 Observation Delays

It has been shown in [19] that DDMDP with no action delays $< S, A, P_A, g, o, 0, c >$ is reducible to an MDP without delays, $< I_o, A, P_A, g' >$, with $I_o = S \times A^o$, where A^o is the Cartesian product of A with itself for o times. Let the state information at the t^{th} time step be contained in $i_t = (s_{t-o}, a_{t-o}, ..., a_{t-1})$, where s_{t-o} is the most recently observed state and $a_{t-o,...}, a_{t-1}$ are the actions taken since. i_t is the expanded system state at time t. Accordingly, the new cost function can be expressed as $g'(i_t, a_t) = E[g(s_t, a_t)|i_t]$. In the equivalent MDP without delays, policies are defined by mappings $\pi : S \times A^o \rightarrow A$. It is also assumed that cost delay is greater than observation delay ($o \leq c$), thus the costs induced after $(k - o)-$ stages have not been collected by the decision maker[19]. Under this assumption, cost delays can be excluded from the definition of i_t. Following this idea, we will show that Q-Learning defined in the expanded space i can be used to attain the optimal policies for DDMDP.

Let us begin with the assumption that the system is at state s, for which an action a is taken. From the agent's perspective, information regarding the current state is to be extracted from the most recently observed state, s_{-o}, and the actions taken since, $a_{-o}, ..., a_{-1}$. Thus, the objective of the revised Q-Learning algorithm is to maximize the expected rewards given that the state s is unknown but can be derived in probability via the sequence $i = (s_{-o}, a_{-o}, ..., a_{-1})$. In other words, an optimal policy is the one that achieves the highest expected return under uncertainty in state s given all the information enclosed in i. $Q(s, a|i)$ denote the value function of state-action pairs (s, a) under the condition that i is observed by the agent (note that s is unknown and random). Formally stated,

$$\pi(i) = \max_a E_s \left(Q^* (s, a|i) \right), \tag{3.3}$$

where E_s denotes the expectation over all possible states given that i is available to the agent. In a MDP with unknown rewards and transition probabilities, the randomness of the equation

above is interpreted two folds. First, the current system state s is stochastic and second, given a fixed state s and action a, the optimal state-action value is approximated iteratively using a stochastic approximation process, such as Q-Learning. Let V_o and Q_o denote the state and action value function with observation delay o.Consequently, in [19], the total (discounted) return under policy π is defined as:

$$V_o^{\pi}(i) = E_{\pi}\left\{\sum_{l=0}^{\infty}\gamma^l g(s_{l-o}, a_{l-o})|i\right\}. \tag{3.4}$$

Likewise, the action value under policy π is given by

$$Q_o^{\pi}(i,a) = g(s,a|i) + E_{\pi}\left\{\sum_{l=1}^{\infty}\gamma^l g(s_{l-o}, a_{l-o})|i\right\}, \tag{3.5}$$

where $g(s,a|i)$ is the expected cost induced by the state-action pair (s,a) given that i is available to the agent (note state s is unknown and random). It has further been shown that the optimal policy is $\arg\max_{\pi} V_o^{\pi}(i)$. Utilizing the Bellman equation for a given i, an optimal policy may be expressed as

$$\pi(i) = \arg\max_{a} Q_o^{\pi}(i,a), \tag{3.6}$$

allowing for the following formulation of the Q-Learning algorithm in the expanded state space i:

$$Q_{t+1}(i,a) = (1-\alpha_{ia}(t))Q_t(i,a) + \alpha_{ia}(t)[r_t(i,a) + \gamma \max_{a'} Q_t(i',a')], \tag{3.7}$$

where $r_t(i,a)$ is the instantaneous reward for action a at time step t given that the system is in state s, which is random from the perspective of the agent. The subsequent state is $i' = (s_{-o+1}, a_{-o+1}, ..., a)$, with $\alpha_{ia}(t)$,denoting the learning rate. Thus,

$$\begin{aligned} r_t(i,a) &= r_t(s,a|i) \\ &= r_t(s,a|s_{-o,}a_{-o}, ..., a_{-1}). \end{aligned} \tag{3.8}$$

Let $(P_a)_{ss'} = P_{ss'}^a$ be the transition probability from state s to state s' upon taking action a. From the Markovian property, the probability of the current state being s given that i is

available to the agent is

$$\begin{aligned}\Pr\{s|i\} &= \Pr\{s|s_{-o}, a_{-o}, a_{-o+1}, ...a_{-1}\} \\ &= \left(P_{a_{-o}} P_{a_{-o+1}} ... P_{a_{-1}}\right)_{s_{-o}s}. \end{aligned} \tag{3.9}$$

The latter represents the $o-$step transition probability from state s_{-o} to state s under a series of actions $\{a_{-o}, a_{-o+1}, ...a_{-1}\}$. Therefore, the expected reward for $r_t(i,a)$ is

$$\begin{aligned} E\left(r_t\left(i,a\right)\right) &= g(i,a) \\ &= E_s\left(g(s,a|i)\right) \\ &= \sum_s \Pr(s|i) \cdot g\left(s,a\right) \\ &= \sum_s \left(P_{a_{-o}} P_{a_{-o+1}} ... P_{a_{-1}}\right)_{s_{-o}s} \cdot g\left(s,a\right), \end{aligned} \tag{3.10}$$

where $g(s,a)$ is the reward function for the corresponding MDP without delays, $< S, A, P_A, g >$.

Next, consider the bootstrapping process. The subsequent state i' given $i = (s_{-o}, a_{-o}, ..., a_{-1})$ and a is $i' = (s_{-o+1}, a_{-o+1}, ..., a)$. Hence, the only new component in i' is the observed state s_{-o+1} at the next step, which is unknown to the agent. Moreover,

$$\Pr\{i'|i\} = \Pr\{s_{-o+1}|s_{-o}, a_{-o}\} = P^{a_{-o}}_{s_{-o}s_{-o+1}}. \tag{3.11}$$

The expected value of $\gamma \max\limits_{a'} Q_t(i', a')$ is thus:

$$\begin{aligned} E\left(\gamma \max_{a'} Q_t(i', a')\right) &= \gamma \left\{ \sum_{i'} \Pr\{i'|i\} \cdot \max_{a'} Q_t\left(i', a'\right) \right\} \\ &= \gamma \left\{ \sum_{s_{-o+1}} \Pr\{s_{-o+1}|s_{-o}, a_{-o}\} \cdot \max_{a'} Q_t(i', a') \right\} \\ &= \gamma \left\{ \sum_{s_{-o+1}} P^{a_{-o}}_{s_{-o}s_{-o+1}} \cdot \max_{a'} Q_t(s_{-o+1}, a_{-o+1}, ..., a') \right\}. \end{aligned} \tag{3.12}$$

Let $T(\cdot)$ be defined as the fixed-point of $Q-$learning process,

$$\begin{aligned} T_{i,a}(Q) &= g(i,a) + \gamma \left\{ \sum_{s_{-o+1}} \Pr\{s_{-o+1}|s_{-o}, a_{-o}\} \cdot \max_{a'} Q^*(s_{-o+1}, a_{-o+1}, ..., a') \right\} \\ &= \sum_{s} \left(P_{a_{-o}} P_{a_{-o+1}} ... P_{a_{-1}}\right)_{s_{-o}s} \cdot g(s,a) \\ &\quad + \gamma \left\{ \sum_{s_{-o+1}} \Pr\{s_{-o+1}|s_{-o}, a_{-o}\} \cdot \max_{a'} Q^*(s_{-o+1}, a_{-o+1}, ..., a') \right\}. \end{aligned} \tag{3.13}$$

Before describing the details of our results, we briefly reiterate the main convergence result for Q-Learning provided in [28].

Theorem 1 *Let $F(t)$ denote all the previous history up to time t and $\tau_{ia}(t)$ the total number of times (i,a) has been visited until t. Q-Learning in the form*

$$Q_{t+1}(i,a) = (1 - \alpha_{ia}(t)) Q_t(i,a) + \alpha_{ia}(t)[w_{ia}(t) + T_{ia}(Q_t)]$$

is convergent given the following assumptions:

1. *For all i and a, $\lim_{t\to\infty} \tau_{ia}(t) = \infty$ w.p.1;*

2. *(a) $Q(0)$ is $F(0)$- measurable;*

   (b) For every i and t, $w_{ia}(t)$ is $F(t+1)-$measurable;

   (c) For every i ,a and $t, \alpha_{ia}(t)$ and $\tau_{ia}(t)$are $F(t)-$measurable;

   (d) For every i, a and t, we have $E(w_{ia}(t)|F(t)) = 0$;

   (e) There exist constants A and B s.t

$$E\left(w_{ia}^2(t)|F(t)\right) \leq A + B \max_{j,v,\tau\leq t} |Q_{jv}(\tau)|^2, \qquad \text{for all } i,a,t.$$

3. *(a) For every i, a,*

$$\sum_{t=0}^{\infty} \alpha_{ia}(t) = \infty, \qquad w.p.1 \text{ for all } i,a.$$

(b) There exist a constant C such that for every i, a,

$$\sum_{t=0}^{\infty} \alpha_{ia}^2(t) \leq C, \quad w.p.1 \text{ for all } i, a.$$

4. There exists a vector x^*, a positive vector v, and a scalar $\beta \in [0,1]$ such that

$$\|T(x) - x^*\|_v \leq \beta \|x - x^*\|_v .$$

Using the above assertions, we state the following:

Theorem 2 *For a Q-Learning algorithm with constant observation delay o, $Q_t(i,a)$ converges to $T_{ia}(Q)$ as defined in (3.13) with probability 1, for every i and a, if the following assumptions are satisfied:*

1. $\gamma < 1$;

2. $\sum_t \alpha_t = \infty$; $\sum_t \alpha_t^2 < \infty$

Proof. From stochastic approximation theory, $T_{i,a}(Q)$ is the optimal value of $Q(i,a)$ if $Q_t(i,a)$ converges as $t \to \infty$. To that end, we next prove the convergence of $Q_t(i,a)$. Given that $T_{i,a}(Q_t) = g(i,a) + \gamma \left\{ \sum_{s_{-o+1}} \Pr\{s_{-o+1} | s_{-o}, a_{-o}\} \cdot \max_{a'} Q_t(i', a') \right\}$, the Q-Learning algorithm can be restated as:

$$Q_{t+1}(i,a) = (1 - \alpha_{ia}(t)) Q_t(i,a) + \alpha_{ia}(t)[w_{ia}(t) + T_{ia}(Q_t)], \tag{3.14}$$

where

$$w_{ia}(t) = r_t(i,a) + \gamma \max_{a'} Q_t(i', a') - T_{ia}(Q_t). \tag{3.15}$$

In the following sections, we select proper values for the discounting factor γ and learning rate α so that the two assumptions in Theorem 2 are inherently satisfied. From Theorem 1, the assumptions 1, 2(a),(b),(c) and 3 are guaranteed in Q-Learning with delays as in the case without delay. In order to assert convergence, we need to show that $E(w_{ia}(t)) = 0$, $E\left(w_{ia}^2(t) | F(t)\right) \leq A + B \max_{j,v,\tau \leq t} |Q_{jv}(\tau)|^2$ and $T_{id}(Q)$ is a contraction mapping with respect

to some norm. From the definition of $T_{ia}(Q_t)$, it directly follows that $E(w_{ia}(t)) = 0$ and $E\left(w_{ia}^2(t)|F(t)\right) \leq Var\left(r_{ia}\right) + \max_{j,v} Q_t^2(j,v)$ as in all the following cases [28]. Next, we show that $T_{id}(\cdot)$ is a contraction mapping with respect to the some norm of $Q(i,a)$, using the following derivation

$$\begin{aligned}
\|T_{ia}(Q_t) - T_{ia}(Q^*)\| &= \Bigg\| g(i,a) + \gamma E\left(\gamma \max_{a'} Q_t(i',a')\right) - g(i,a) - \\
&\quad \gamma E\left(\gamma \max_{a'} Q^*(i',a')\right) \Bigg\| \\
&= \Bigg\| \gamma \sum_{s_{-o+1}} P^{a_{-o}}_{s_{-o}s_{-o+1}} \max_{a'} Q_t(i',a') - \\
&\quad \gamma \sum_{s_{-o+1}} P^{a_{-o}}_{s_{-o}s_{-o+1}} \max_{a'} Q^*(i',a') \Bigg\| \\
&\leq \gamma \sum_{s_{-o+1}} P^{a_{-o}}_{s_{-o}s_{-o+1}} \left\| \max_{a'} Q_t(i',a') - \max_{a'} Q^*(i',a') \right\| \\
&\leq \gamma \sum_{s_{-o+1}} P^{a_{-o}}_{s_{-o}s_{-o+1}} \max_{a'} \left\| Q_t(i',a') - Q^*(i',a') \right\| \\
&\leq \gamma \max_{i'} \left\{ \max_{a'} \left\| Q_t(i',a') - Q^*(i',a') \right\| \right\} \\
&\leq \gamma \max_{j,v} \left\| Q_t(j,v) - Q^*(j,v) \right\| . \qquad (3.16)
\end{aligned}$$

Given the above, $T_{ia}(\cdot)$ is a contraction mapping with respect to the sup norm, which concludes the proof. ■

3.3.2 Action Delays

In [18], it has been shown that the effects of observation and action delays on the structure of the equivalent MDP without delays are additive, such that the DDMDP $< S, A, P_A, g, o, ac, c >$ is reducible to the MDP $< I_{o+ac}, A, P_A, g_{o+ac} >$. Given the system state and the action issued by the agent, the additional information relevant for the subsequent decision making comprises of the actions issued since the time of the most recently observed state. Utilizing such information, the distribution of system states s_{ac}, at which action a will take effect, can be evaluated. Thus, the state is given by $i = (a_{-o-ac}, ..., a_{-o-1}, s_{-o}, a_{-o}, ..., a_{-1})$. Following similar rationale to that

which has led to (3.14), the Q-Learning algorithm can be formulated as:

$$Q_{t+1}(i,a) = (1-\alpha_{ia}(t))Q_t(i,a) + \alpha_{ia}(t)[r_t(i,a) + \gamma \max_{a'} Q_t(i',a')]. \tag{3.17}$$

The difference between the two cases lies in the reward signal $r_t(i,a)$. In the case of action delays, the action a takes effect not at state s, but rather at state s_{ac}, since it is delayed by ac stages. Consequently, we may write

$$\begin{aligned} r_t(i,a) &= r_t\left(s_{ac}, a | i\right) \\ &= r_t\left(s_{ac}, a | a_{-o-ac}, ..., a_{-o-1}, s_{-o}, a_{-o}, ..., a_{-1}\right). \end{aligned} \tag{3.18}$$

Notice that a_{-o-ac} takes effect at state s_{-o}, and so forth. The distribution of state s_{ac} is

$$\Pr\{s_{ac}|i\} = \left(P_{a_{-o-ac}} P_{a_{-o-ac+1}} ... P_{a_{-1}}\right)_{s_{-o} s_{ac}}, \tag{3.19}$$

reflecting on the $ac+o-1$ step transition probability from state s_{-o} to state s_{ac}. The expected reward is given by

$$\begin{aligned} E\left(r_t\left(i,a\right)\right) &= g(i,a) \\ &= \sum_{s_{ac}} \left(P_{a_{-o-ac}} P_{a_{-o-ac+1}} ... P_{a_{-1}}\right)_{s_{-o} s_{ac}} \cdot g\left(s_{ac}, a\right). \end{aligned} \tag{3.20}$$

Likewise, the subsequent state is $i' = \left(a_{-o-ac+1}, ..., a_{-o-1}, s_{-o+1}, a_{-o}, ..., a\right)$, which leads to the transition probability

$$\Pr\{i'|i\} = \Pr\{s_{-o+1}|s_{-o}, a_{-o-a}\} = P^{a_{-o-ac}}_{s_{-o} s_{-o+1}}. \tag{3.21}$$

The discounted expected value of the subsequent state is

$$\begin{aligned} &E\left(\gamma \max_{a'} Q_t(i',a')\right) = \\ &\gamma \left\{ \sum_{s_{-o+1}} P^{a_{-o-ac}}_{s_{-o} s_{-o+1}} \cdot \max_{a'} Q_t\left(a_{-o-ac+1}, ..., a_{-o-1}, s_{-o+1}, a_{-o}, ..., a'\right) \right\}. \end{aligned} \tag{3.22}$$

We formulate $T_{ia}(Q)$ as:

$$\begin{aligned} T_{i,a}(Q) &= g(i,a) + \gamma \left\{ \sum_{s_{-o+1}} \Pr\{s_{-o+1}|s_{-o}, a_{-o}\} \cdot \max_{a'} Q^*(i', a') \right\} \\ &= \sum_{s_{ac}} \left(P_{a_{-o-ac}} P_{a_{-o-ac+1}} ... P_{a_{-1}} \right)_{s_{-o} s_{ac}} \cdot g(s_{ac}, a) \\ &\quad + \gamma \left\{ \sum_{s_{-o+1}} P^{a_{-o-ac}}_{s_{-o} s_{-o+1}} \cdot \max_{a'} Q_t \left(a_{-o-ac+1}, ..., a_{-o-1}, s_{-o+1}, a_{-o}, ..., a' \right) \right\}. \end{aligned} \quad (3.23)$$

Theorem 3 *For a Q-Learning algorithm with constant observation delay o and constant action delay ac, $Q_t(i,a)$ converges to $T_{ia}(Q)$ in (3.23) with probability 1, for every i and a, given that the two assumptions in Theorem 2 are satisfied.*

Proof. Here $T_{ia}(\cdot)$ is almost identical to that referred to in the case of no action delays, with the exception of subscript differences for the state and action. Substituting o with $o+a$, it can be easily shown that $T_{ia}(\cdot)$ is a contraction mapping with regard to the sup norm, proving that Q-Learning with observation delays and action delays converge similarly. ■

3.4 Random Delays

3.4.1 No Action Delays

We next discuss stochastic delayed MDPs (SDMDPs) $< S, A, P_A, g, O, 0, C, >$, with the random variables O denoting observation delays and C denoting the delay in collecting rewards. Let the current observation delay be defined as $o \in O$, where the latter denotes the sample space of O. Let the observation delay at the subsequent stage be $o' \in O$. We assume that for all time steps t, it is possible to observe state s_{t+1} only after state s_t has been observed. In another words, the observation delay at the current stage is always less or equal to that of the following stage, i.e. $\Pr\{o \leq o'\} = 1$. We also define an upper bound on the observation delay of $o_{\max}$, such that $\Pr\{O \leq o_{\max}\} = 1$. When this upper bound is reached, it is assumed that the decision-making process freezes in the sense that it takes no actions until the most recent system state is observed. During this time period no new costs are induced since no actions are taken. However, although the underlying system does not make any new state transitions during that

time period, the agent does continue to observe the previous system state transitions. This is a plausible assumption in the sense that making decisions with very old state information is highly undesirable. Moreover, it keeps the state vector dimension from increasing to infinity [18].

The state observed by the agent in this case is $i = (s_{-o}, a_{-o}, ...a_{-1})$. It should be noted that here o is a possible value of the random variable O. The subsequent state of the agent, i', can take two values. If $o < o'$, no new state is observed, therefore $i'_{o<o'} = (s_{-o}, a_{-o}, ...a)$. However, if $o = o'$, the observation delays at the two stages are identical, which means that the most recently observed state at the subsequent step will shift to s_{-o+1}, yielding $i'_{o=o'} = (s_{-o+1}, a_{-o+1}, ..., a)$ .So, if O is geometrically distributed with parameter p conditioned on $\Pr\{o \leq o'\} = 1$, it is easy to show that $\Pr\{o < o'\} = 1 - p$ and $\Pr\{o = o'\} = p$. Hence, for a given state $i = (s_{-o}, a_{-o}, ..., a_{-1})$ and action a, the Q-Learning algorithm with random observation delays $o \in O$ can be formulated, similarly to (3.14), as

$$Q_{t+1}(i,a) = (1 - \alpha_{ia}(t))Q_t(i,a) + \alpha_{ia}(t)[r_t(i,a) + \gamma \max_{a'} Q_t(i',a')],$$

where i' is the subsequent state $i'_{o<o'} = (s_{-o}, a_{-o}, ...a)$ or $i'_{o=o'} = (s_{-o+1}, a_{-o+1}, ..., a)$. The expected reward for $r_t(i,a)$ is

$$E\left(r_t(i,a)\right) = g\left(i,a\right) = \sum_{s} \left(P_{a_{-o}} P_{a_{-o+1}} ... P_{a_{-1}}\right)_{s_{-o}s} \cdot g(s,a). \tag{3.24}$$

The expected value of the subsequent state $\gamma \max_{a'} Q_t(i',a')$ is

$$\begin{aligned} E\left(\gamma \max_{a'} Q_t\left(i',a'\right)\right) &= \\ &\gamma \left\{ P(o < o') E\left(\max_{a'} Q_t\left(i'_{o<o'}, a'\right)\right) + \right. \\ &\left. P(o = o') E\left(\max_{a'} Q_t\left(i'_{o=o'}, a'\right)\right) \right\} \\ &= \gamma \left\{ P(o < o') \max_{a'} Q_t\left(i'_{o<o'}, a'\right) \right. \end{aligned}$$

$$
\begin{aligned}
&+P(o=o')\sum_{i_{o=o'}}\Pr\{i_{o=o'}|i\}\max_{a'}Q_t(i'_{o=o'},a')\Bigg\}\\
=\;&\gamma\Bigg\{P(o<o')\max_{a'}Q_t\left(s_{-o},a_{-o},...a'\right)\\
&+P(o=o')\sum_{s_{-o+1}}P^{a_{-o}}_{s_{-o}s_{-o+1}}\max_{a'}Q_t(s_{-o+1},a_{-o+1},...,a')\Bigg\},
\end{aligned}
\tag{3.25}
$$

and the optimal value is

$$
\begin{aligned}
&T_{i,a}\left(Q\right)=\sum_{s}\left(P_{a_{-o}}P_{a_{-o+1}}...P_{a_{-1}}\right)_{s_{-o}s}\cdot g\left(s,a\right)\\
&+\gamma P\left(o<o'\right)\max_{a'}Q^{*}\left(s_{-o},a_{-o},...a'\right)+\\
&P\left(o=o'\right)\sum_{s_{-o+1}}P^{a_{-o}}_{s_{-o}s_{-o+1}}\max_{a'}Q^{*}(s_{-o+1},a_{-o+1},...,a').
\end{aligned}
\tag{3.26}
$$

Theorem 4 *For a Q-Learning algorithm with random observation delay O, $Q_t(i,a)$ converges to $T_{ia}(Q)$ in (3.26) with probability 1, for every i and a, and possible observation delay $o\in O$, given that the two assumptions in Theorem 2 are satisfied.*

Proof. Letting

$$
\begin{aligned}
T_{i,a}(Q_t)=g(i,a)+\gamma\Bigg\{&P(o<o')\max_{a'}Q_t(i_{o<o'},a')\\
&+P\left(o=o'\right)\sum_{s_{-o+1}}P^{a_{-o}}_{s_{-o}s_{-o+1}}\max_{a'}Q_t(i'_{o=o'},a')\Bigg\},
\end{aligned}
\tag{3.27}
$$

we show that $T_{ia}(\cdot)$ is a contraction mapping,

$$
\begin{aligned}
&\|T_{ia}(Q_t)-T_{ia}(Q^*)\|=\\
&\left\|g(i,a)+\gamma E\left(\max_{a'}Q_t(i',a')\right)-g(i,a)-\gamma E\left(\max_{a'}Q^*(i',a')\right)\right\|\\
=\;&\gamma P(o<o')\left\|\max_{a'}Q_t(i_{o<o'},a')-\max_{a'}Q^*(i_{o<o'},a')\right\|+\gamma P\left(o=o'\right)\cdot\\
&\left\|\sum_{s_{-o+1}}P^{a_{-o}}_{s_{-o}s_{-o+1}}\max_{a'}Q_t(i_{o=o'},a')-\sum_{s_{-o+1}}P^{a_{-o}}_{s_{-o}s_{-o+1}}\max_{a'}Q^*(i_{o=o'},a')\right\|
\end{aligned}
$$

$$
\begin{aligned}
&\leq \gamma P(o<o')\max_{a'}\left\|Q_t\left(i_{o<o'},a'\right)-Q^*\left(i_{o<o'},a'\right)\right\| \\
&\quad +\gamma P\left(o=o'\right)\max_{i_{o=o'},a'}\left\|Q_t\left(i_{o=o'},a'\right)-Q^*\left(i_{o=o'},a'\right)\right\| \\
&\leq \gamma\left\{P(o<o')+P\left(o=o'\right)\right\}\max_{j,v}\left\|Q_t(j,v)-Q^*(j,v)\right\| \\
&\leq \gamma\max_{j,v}\left\|Q_t(j,v)-Q^*(j,v)\right\|. \qquad (3.28)
\end{aligned}
$$

From the above, it follows that $T_{ia}(\cdot)$ is a contraction mapping with regard to the sup-norm, therefore, Q-Learning with random observation delays converges. ■

3.4.2 Action Delays

We define the SDMDP with observation and action delays as the seven-tuple $< S, A, P_A, g, O, AC, C >$. For notation purposes, we let the current observation delay be denoted by o, and label the action delay at stage $-o$ as ac. That is, the action a_{-o-ac} takes effects when the system state is s_{-o}. Correspondingly, the observation delay at the subsequent stage is o'. Likewise, we assume that observations are ordered, such that $\Pr(o \leq o') = 1$. Note that $o, o' \in O$ and $ac \in AC$ are samples from the sample spaces of O and AC, respectively. The objective of Q-Learning is to maximize the expected return for some future state since the action is delayed. Intuitively, the distribution of possible future states, where a will take effect, can be inferred by the most recently observed state and all actions taken up to the previous one. Therefore, the state of the agent is $i = (a_{-o-ac}, ..., a_{-o-1}, s_{-o}, a_{-o}, ..., a_{-1})$. We denote the future (uncertain) state by s. For a given action a, the expected instant reward is

$$
E\left(r_t(i,a)\right) = g\left(i,a\right) = \sum_{s}\left(P_{a_{-o-ac}}P_{a_{-o-ac+1}}...P_{a_{-1}}\right)_{s_{-o}s}\cdot g\left(s,a\right). \qquad (3.29)
$$

Repeating the arguments stated above, the subsequent state depends on whether the next observation s_{-o+1} is available or not. As such, the optimal value is given by

$$
\begin{aligned}
T_{i,a}\left(Q\right) &= \sum_{s}\left(P_{a_{-o-ac}}P_{a_{-o-ac+1}}...P_{a_{-1}}\right)_{s_{-o}s}\cdot g\left(s,a\right) \\
&\quad +\gamma P\left(o<o'\right)\max_{a'}Q^*\left(a_{-o-ac},...,s_{-o},a_{-o}...a'\right) \\
&\quad +P\left(o=o'\right)\sum_{s_{-o+1}}P^{a_{-o-ac}}_{s_{-o}s_{-o+1}}\max_{a'}Q^*(a_{-o-ac+1},...,s_{-o+1},a_{-o}...,a'). \qquad (3.30)
\end{aligned}
$$

$Q-$**learning with Delays**
1. Given:
(a) an ergodic MDP with state space S,observation delay space O,and bounded reward;
2. For $t = 1$ to ∞ :
(a) Interact with POMDP:
1) given observation of state delayed by o, s_{t-o}, and the actions afterward expanded state is $i_t = \{s_{t-o}, a_{t-o}, a_{t-o}, ..., a_{t-1}\}$.
4) take action a_t: $a_t = \pi(i_t)$.
5) observe the reward r_{t+1};
(b) Update the Q-value if next delayed state s_{t-o+1} is available
1) derive expand next state as $i_{t+1} = \{s_{t-o+1}, a_{t-o+1}, ..., a_{t-1}, a_t\}$
2) update Q-table, α is the learning rate and γ discounting factor $Q_{t+1}(i_t, a_t) = (1-\alpha)Q_t(i_t, a_t) + \alpha[r_{t+1} + \gamma \max_{a'} Q_t(i_{t+1}, a')]$,
(c) $i_t = i_{t+1}$

Table 3.1: Q-Learning with delays.

Theorem 5 *For a Q-Learning algorithm with random observation delay O and random action delay AC, $Q_t(i, a)$ converges to $T_{ia}(Q)$ in (3.30) with probability 1, for every i and a and possible observation and action delay pair ($o \in O, ac \in AC$), given that the two assumptions in Theorem 2 are satisfied.*

Proof. The proof can be easily obtained by following the same derivations as in Theorems 2 and 3. ■

3.5 Algorithm Outline for Q-Learning with Delays

The algorithm for $Q-$learning with delays is listed in Table 3, assume that the agent records all its action history. Further, we only consider observation delays here because action delays and observation delays are interchangeable 3.3.2

Form the algorithm in 3.1, the storage requirement of $Q-$learning on the augmented state space i is increased exponentially with the upper bound of observation delay, $\max\{O\}$, which determines the dimension of the state vectors. Further, the augmented space is much sparser, resulting in a very slow convergence rate. Thus, tabular form $Q-$learning is not applicable in the scenario. Next, we will explore function approximation and neurodynamic programming techniques to solve RL problems with high dimensional or continuous state and action space.

Chapter 4

Scalable, Real-time NeuroDynamic Programming (NDP)

4.1 Truncated Real-Time Recurrent Learning (TRTRL)

In this section, we present TRTRL - a resource-efficient variant of RTRL. The notations used here are identical to those in section 2.1.3. TRTRL aims to overcome the inherent scalability limitations of RTRL while retaining its key performance attributes. It accomplishes this goal by reducing the amount of resources required for each neuron. Let us begin with several key definitions that would guide us through the discussion:

Definition 6 *Let I_j denote the set of nodes that have a direct link (and, hence, a unique associated weight) to node j. We shall refer to this set as the ingress set of node j.*

Definition 7 *Let E_j denote the set of nodes that node j has a link (and, hence, a unique associated weight) to. We shall refer to this set as the egress set of node j.*

It should be observed that a node can reside within both ingress and egress sets of another node. Given that TRTRL limits the sensitivities of each neuron to the ingress and egress set, we have the following slightly-revised definition for $z_j(t)$,

$$z_j(t) = \begin{cases} x_j(t) & \text{if } j \in I \\ y_j(t) & \text{if } j \in I_j \end{cases}, \tag{4.1}$$

where I denotes the set of external inputs. By the same token as RTRL, we have:

$$y_k(t+1) = f_k\left(s_k\left(t\right)\right), \tag{4.2}$$

where

$$s_k(t) = \sum_{l \in N \cup M} w_{kl} z_l(t), \tag{4.3}$$

and the error function:

$$J(t) = \frac{1}{2} \sum_{k \in \text{outputs}} \left[d_k(t) - y_k(t)\right]^2 = \frac{1}{2} \sum_{k \in \text{outputs}} \left[e_k(t)\right]^2, \tag{4.4}$$

where $d_k(t)$ denotes the desired target value for output k at time t. Correspondingly, negative gradient direction is:

$$-\frac{\partial J(t)}{\partial w_{ij}(t)} = \sum_{k \in \text{outputs}} e_k(t) \frac{\partial y_k(t)}{\partial w_{ij}} = \sum_{k \in \text{outputs}} e_k(t) p_{ij}^k(t), \tag{4.5}$$

and the updating is given,

$$w_{ij}(t+1) = w_{ij}(t) + \alpha \sum_{k \in \text{outputs}} e_k(t) p_{ij}^k(t).$$

However, in RTRL, the sensitivities are calculated as:

$$p_{ij}^k(t+1) = f_k'\left(s_k\left(t\right)\right) \left[\sum_{l \in N} w_{kl} p_{ij}^l(t) + \delta_{ik} z_j(t)\right]. \tag{4.6}$$

By localizing the information required by each neuron, the calculation of equation 4.6 is constructed of three main parts. First, for all nodes that are not in the output set, the egress sensitivity values for node j are calculated such that $j = k$ thereby yielding the following reduced expression:

$$p_{ij}^j(t+1) = f_j'\left(s_j\left(t\right)\right) \left[w_{ji} p_{ij}^i(t) + \delta_{ij} y_j(t)\right]. \tag{4.7}$$

Notice that the summation from equation 4.6 drops out because $p_{ik}^l = 0$ unless $l = i$. The sensitivities pertaining to the ingress set for node j are calculated where $i = k$ and can be

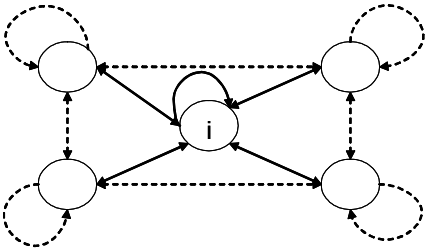

Figure 4-1: The sensitive weights of the ith node.

expressed as

$$p_{ij}^{i}(t+1) = f_i'\left(s_i\left(t\right)\right)\left[w_{ij}p_{ij}^{j}(t) + z_j(t)\right]. \tag{4.8}$$

From the above two expressions it becomes evident that the aggregate computational load for each neuron is in the order of $2N$. Fig 4-1 shows the sensitive weights of the i^{th} node, the weights between the i^{th} node and its ingress and egress nodes. The full calculation of equation 4.6 is performed for input, bias and output units. Furthermore, the network remains fully recurrent in the sense that all neurons are connected (via unique weights) to all other neurons. The only difference between TRTRL and RTRL, in this context, is that neurons are limited in the sensitivities. To that end, TRTRL is completely local because neurons are no-longer required to fetch information that may be located at a remote part of the network. If the network is implemented in hardware, localizing the memory access is key to guaranteeing high-speed of execution.

In order to complete the description of TRTRL, we refer to the weight update rule given in (4.6) and (4.5). For the output neurons, a non-zero sensitivity element must exist in order to provide the performance gradient required by the weight update rule. To comply with this requirement, a direct link is added from each output neuron to each of the N neurons in the network. Therefore, each output neuron, o, computes N^2 sensitivity updates (one for each weight in the network) by performing the following:

$$p_{ij}^{o}(t+1) = f_o'\left(s_o\left(t\right)\right)\left[w_{oi}p_{ij}^{i}(t) + w_{oj}p_{ij}^{j}(t) + \delta_{io}z_j(t)\right]. \tag{4.9}$$

The recursive calculation of sensitivity elements for output neurons is reduced to the two left-most terms in the expression above since only two neurons (i and j) are sensitive to w_{ij}. This yields an overall computational complexity of $O(KN^2)$, where K denotes the number of output neurons in the network. Moreover, storage complexity is $O(N^2)$.

4.1.1 SMD for TRTRL

In this section, we first review the stochastic meta-descent (SMD) algorithm, first introduced in [29]. As an alternative to utilizing small, identical constant learning rates for all network weight updates, SMD employs an independent learning rate for each weight. Accordingly, the weight update rule is given by

$$w_{ij}(t+1) = w_{ij}(t) + \lambda_{ij}(t)\delta_{ij}(t), \tag{4.10}$$

where $\lambda_{ij}(t)$ is the learning rate for weight w_{ij} at time t. Moreover, the local learning rates are independently adapted by exponentiated gradient descent. In this way, they can cover a wide dynamic range while remaining strictly positive [30]. Accordingly, the following learning rate update rule is used:

$$\ln \lambda_{ij}(t) = \ln \lambda_{ij}(t-1) - \mu \frac{\partial J(t)}{\partial \ln \lambda_{ij}}, \tag{4.11}$$

where μ is a global meta-learning rate. Using the chain rule, the above can be rewritten as

$$\begin{aligned} \ln \lambda_{ij}(t) &= \ln \lambda_{ij}(t-1) - \mu \frac{\partial J(t)}{\partial w_{ij}(t)} \frac{\partial w_{ij}(t)}{\partial \ln \lambda_{ij}} \\ &= \ln \lambda_{ij}(t-1) + \mu \delta_{ij}(t) v_{ij}(t) \end{aligned} \tag{4.12}$$

where

$$v_{ij}(t) = \frac{\partial w_{ij}(t)}{\partial \ln \lambda_{ij}}. \tag{4.13}$$

This approach rests on the assumption that each element of λ affects J only through the corresponding element of w. To avoid an expensive exponentiation for each weight update, eq. (4.12) is further simplified by exploiting the linearization $e^{\mu} = 1 + \mu$, valid for small $|\mu|$, to yield

$$\lambda_{ij}(t) = \lambda_{ij}(t-1) \max\left(\rho, 1 + \mu \delta_{ij}(t) v_{ij}(t)\right), \tag{4.14}$$

where ρ (typically around 0.5) is a safeguard factor against unreasonably small, or negative, values. Meta-level gradient descent remains stable as long as $\delta_{ij}(t)v_{ij}(t), \forall\ i, j$ does not stray away from unity. Next, v_{ij} is expressed as a gradient trace that measures the long-term impact of a change in a local learning rate to its corresponding weight. Accordingly, the SMD algorithm defines v_{ij} as an exponential average of the effect of all past learning rates on the new weight values, such that

$$v_{ij}(t+1) = \sum_{k=0}^{\infty} \beta^k \frac{\partial w_{ij}(t+1)}{\partial \ln \lambda_{ij}(t-k)}, \tag{4.15}$$

where the coefficient $0 < \beta < 1$ determines the time scale over which long-term dependencies are taken into account. Eq. (4.15) can be effectively approximated to yield the following:

$$v_{ij}(t+1) = \beta v_{ij}(t) + \lambda_{ij}(t) \left(\delta_{ij}(t) - \beta \left(H_t v(t) \right)_{ij} \right), \tag{4.16}$$

where $v_{ij}(0) = 0, \forall i, j$ and H_t denotes the instantaneous Hessian (the matrix of second derivatives $\partial^2 J / \partial w_{ij} w_{kl}$ of the error J with respect to each pair of weights) at time t. The two equations (4.14) and (4.16) complete the updating of the learning rates λ_{ij} for each w_{ij}.

The objective of the TRTRL algorithm, which is essentially an online optimization technique, is to minimize a global error function, J, such that the network's future outputs will be closer to their designated targets. What makes TRTRL and its variants unique is that they are online schemes, whereby each time step an error is provided, based on which the network parameters (i.e. weights) are updated. As such, TRTRL is a *stochastic gradient* based method that aims to optimize the network's performance by utilizing instantaneous gradient information. Network weights are updated iteratively along the negative gradient direction,

$$w_{ij}(t+1) = w_{ij}(t) + \alpha \delta_{ij}(t), \tag{4.17}$$

where

$$\delta_{ij}(t) = -\frac{\partial J(t)}{\partial w_{ij}}, \tag{4.18}$$

and α is the learning rate parameter. In practice, the learning rate α is set to a small constant value in order to guarantee convergence of the training algorithm and avoid oscillations in a direction where the error function is steep. However, this approach considerably slows down

training since, in general, a small learning rate may not be appropriate for all portions of the error surface [31]. To address this issue, stochastic meta-descent (SMD)[29] [32] applies an adjustable learning rate for every connection (weight) in the network, in an attempt to use not only the gradient but also the second derivative of the error function as means of accelerating the learning process.

In applying SMD to TRTRL, the primary task is to derive an efficient algorithm for obtaining $H_t v_t$. At first glance, this might suggest a computationally heavy process. Fortunately this is not the case, since there are very efficient indirect methods for computing the product of the Hessian with an arbitrary vector [33] [34]. To prevent negative eigenvalues from causing (4.14) to diverge, SMD uses an extended Gauss-Newton approximation that also admits a fast matrix-vector product. Pearlmutter [33] presented an exact and numerically stable procedure to compute $H_t v_t$ with a computational complexity of $O(n)$ and no need to explicitly calculate or store the matrix H_t. We begin by reviewing this technique. It has been show that the product of a Hessian H with any arbitrary vector, v, can be computed as

$$Hv = R_v\{\nabla_w\} = \frac{\partial}{\partial r}\nabla_{(w+rv)}\,|_{r=0} \tag{4.19}$$

where $R_v\{\cdot\}$ is a differential operator and r is a real value. ∇_w is the gradient of the optimized function with respect to the adjustable parameters w. rv is considered a small perturbation to ∇_w in the direction of v. In the context of neural networks, ∇_w is the gradient of the error function to the weights and v is the gradient trace defined in eq (4.13). Applying the $R_v\{\cdot\}$ operator to TRTRL, we obtain

$$\begin{aligned} -\left(H_t v(t)\right)_{ij} &= R_v\left\{-\nabla_{w_{ij}}\right\} \\ &= R_v\left\{\delta_{ij}(t)\right\} \\ &= R_v\left\{-\frac{\partial J(t)}{\partial w_{ij}}\right\} \\ &= R_v\left\{\sum_{o \epsilon output} e_o(t) p^o_{ij}(t)\right\} \end{aligned} \tag{4.20}$$

$$
\begin{aligned}
&= \sum_{o \epsilon output} \left[e_o(t) R_v \left\{ p_{ij}^o(t) \right\} + \right. \\
&\quad \left. R_v \left\{ e_o(t) \right\} p_{ij}^o(t) \right] \\
&= \sum_{o \epsilon output} \left[e_o(t) R_v \left\{ p_{ij}^o(t) \right\} - \right. \\
&\quad \left. R_v \left\{ y_o(t) \right\} p_{ij}^o(t) \right]
\end{aligned}
$$

Next, we need to calculate $R_v \{y_o(t)\}$ and $R_v \left\{ p_{ij}^o(t) \right\}$. From the Eq.(4.3), we note that

$$
R_v \{s_o(t)\} = \sum_{l \epsilon U \cup I} v_{ol}(t) z_l(t), \tag{4.21}
$$

such that and from Eq.(4.2),

$$
R_v \{y_o(t)\} = f'(s_o(t)) R_v \{s_o(t)\}. \tag{4.22}
$$

By the same token and Eq.(4.9),

$$
\begin{aligned}
R_v \left\{ p_{ij}^o(t) \right\} &= f'' \left(s_o(t) \right) R_v \left\{ s_o(t) \right\} \\
&\quad \cdot \left[w_{oi} p_{ij}^i(t) + w_{oj} p_{ij}^j(t) + \delta_{io} z_j(t) \right] \\
&\quad + f'(s_o(t)) \left[v_{oi} p_{ij}^i(t) + v_{oj} p_{ij}^j(t) \right]
\end{aligned} \tag{4.23}
$$

Note that the computation of $H_t v_t$ only incurs calculations in the output neurons thus adds only a little to the overall computations. It should also be noted that the calculation of $H_t v_t$ can be thought of as a concurrent and adjoint process to the gradient calculation, with a similar computational complexity of $O(KN^2)$, where K denotes the number of output neurons. Moreover, the storage requirements are still $O(N^2)$.

Adaptation of the Global Meta-learning Rate μ

The original SMD technique does not consider any adaptation of the global meta-learning rate parameter, μ.In fact, the latter is often viewed as the "learning rate of the learning rate", with typical values in the order of 0.1. To ensure faster convergence and stability of the algorithm as a whole, we introduce an adaptive global meta-learning rate by the same heuristic techniques

of SuperSAB [34] [35]. We increase the value of μ if a positive correlation between successive gradients of the error function with respect to learning rate is observed, otherwise μ is decreased. Let φ be the negative gradient of the error function with respect to the exponentiated learning rate such that

$$\varphi_{ij}(t) = -\frac{\partial J(t)}{\partial \ln \lambda_{ij}} = \delta_{ij}(t) v_{ij}(t). \tag{4.24}$$

Accordingly, $\mu_{ij}(t)$ is updated in the following manner:

$$\mu_{ij}(t) = \mu_{ij}(t-1)\left(1 + \eta \varphi_{ij}(t)\varphi_{ij}(t-1)\right), \tag{4.25}$$

where $\eta = .05$ is a small positive constant. Moreover, μ_{ij} is bounded by $[\mu_{\min} = 0.01, \mu_{\max} = 5]$ in order to ensure stability and smoother learning.

4.1.2 Discussion on Storage and Computational Complexity

Primary benefits of TRTRL, from an implementation perspective, are the substantial reductions in computation complexity and storage requirements. Computation time is dominated by the calculation of the sensitivity elements. While in the original RTRL scheme, each neuron required to perform $O(N^3)$ floating-point operations (flops), TRTRL requires only $O(N)$. Note that SMD necessitates approximately three times the flops involved in regular gradient computations. This results in an overall (network-level) computational complexity of $O(N^2)$, instead of $O(N^4)$ that characterizes RTRL.

A similar reduction in resources is observed in the storage requirements of TRTRL. All N^3 elements of the sensitivity matrix are required in RTRL, while TRTRL only operates on $2N$ sensitivities per neuron. As such, the overall storage requirement drops from $O(N^3)$ to $O(N^2)$. It should be noted that, as opposed to RTRL, TRTRL is a highly localized algorithm. This contributes to the more effective implementation prospect of the scheme in hardware. Moreover, it is interesting to note that although this chapter addresses the case of fully-connected networks, the TRTRL formalism is not restricted to such cases. In fact, assuming that each node is only connected to M other nodes, the computational complexity becomes $O(KMN)$ while storage is reduced to $O(MN)$. The only constraint imposed in such cases is that each node have a direct link to the output neurons (as means of propagating error information), as dictated by (4.9).

4.1.3 Performance Analysis

We performed a comparison between the RTRL, TRTRL, TRTRL with SMD (TRTRL-SMD) algorithms for two commonly employed testbenches that require the network to capture temporal dependencies: frequency doubling and chaotic time series prediction. In both cases, information that arrives at a given time has strong impact on the value of outputs at subsequent time steps. In essence, it is a measure of the meaningfulness of neuron activations (which are the "soft" state of the network) in order to successfully accomplish the tasks.

Frequency Doubler

The first task chosen was the frequency doubler system. For this task, the network was required to produce a sinusoidal signal that has twice the frequency of the signal applied at its input. The latter is a sinusoid with a 16-sample period while the desired output signal is a sinusoid with an 8-sample period. This is a suitable basic task for the network as the input to output mapping is nonlinear and requires memory.

For RTRL, TRTRL and TRTRL-SMD, the network consisted of a single hidden layer with 15 fully recurrent neurons, one bias input neuron whose (constant) value is 1 and a single linear output neuron. The same set of random initial weights was applied to the three networks each time they were trained. Moreover, an initial learning rate of $\lambda_{ij}(0) = 0.01, \forall\ i, j$ applied to all three algorithms, with TRTRL-SMD gradually adapting its learning rate to accelerate the learning process. The TRTRL-SMD algorithm parameters were configured with the following initial values $\rho = 0.5, \beta = 0.95, \mu_{ij}(0) = 0.1$, where $\mu_{ij} \in [0.01, 5]$, and $\eta = 0.05$.

Figure 4-2 below depicts the average learning curves for the three algorithms over 20 runs. While TRTRL converges a little slower than RTRL, SMD improves the learning rate of TRTRL to a level that surpasses the performance of RTRL. It further appears that, as the iteration count increases, the performance advantage of TRTRL-SMD also increases. This serves as a basic indication that, despite the partial sensitivities inherent to TRTRL, the gradient-based information propagated through the weights and activations of the network is sufficient to provide meaningful modeling capabilities.

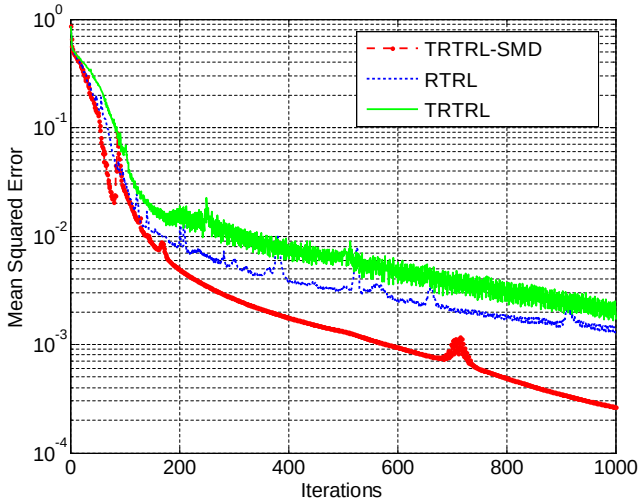

Figure 4-2: Average learning curves for the frequency doubler testbench, comparing a 15-neuron fully-recurrent network running RTRL, TRTRL and TRTRL/SMD.

Chaotic Time Series Prediction

The next task chosen was chaotic time series prediction, whereby the networks are required to predict future values of the Mackey-Glass (MG) [36] [37] chaotic series, which has been extensively used as a benchmark. The MG series is based on the time-delayed differential equations,

$$\frac{\partial x(t)}{\partial t} = -0.1x(t) + \frac{0.2x(t-\tau)}{1+x^{10}(t-\tau)} \tag{4.26}$$

To obtain values at integer time points, the fourth-order Runge-Kutta method was used to find the numerical solution to the above MG equation. Here, we assume that the time step is 1, $x(0) = 0.1, \tau = 17$ and $x(t) = 0$ for $t = 0$.

The task is to predict the value of $x(t+30)$ given the current input and internal state representation. The chaotic time series prediction task was chosen because it is significantly more difficult for the networks to solve than the frequency doubler. The network topology was identical to the one used for the frequency doubler task. The three networks were constructed using 25 fully recurrent neurons, one bias input neuron whose (constant) value is 1 and a single linear output neuron. The same set of random initial weights was used for each network during

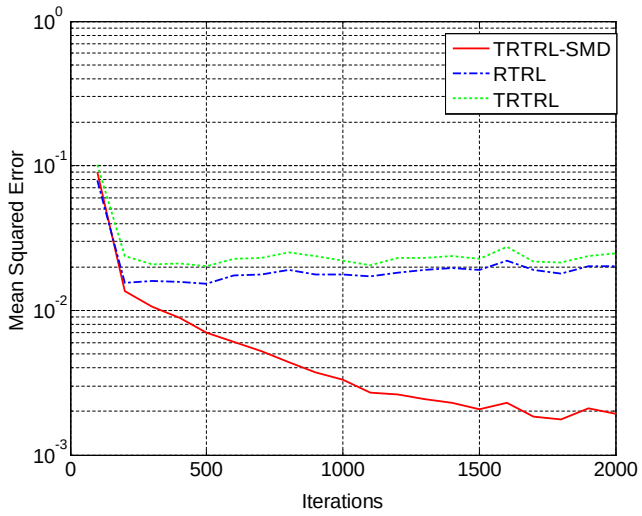

Figure 4-3: Learning curves for the chaotic time series prediction task, applied to a 25-neuron network running TRTRL-SMD, RTRL and TRTRL

every training run. The same settings of parameters for the three algorithm are used in this test as in the frequency doubler testbench.

Figure 4-3 illustrates the average learning curves for the three algorithm over 20 runs. This simulation task proved difficult for both RTRL and TRTRL as the error did not drop below 10^{-2}, with RTRL demonstrating a slight advantage over TRTRL. In contrast, TRTRL-SMD has a clearly higher convergence rate, and higher degree of accuracy. It is the tendency of SMD to effectively adapt the step size in narrow error hyper-surfaces that is attributed the substantial improvement in performance on this test case.

4.2 Clustered TRTRL

In order to improve the TRTRL algorithm for hardware scalability, the number of connections between neurons should be further reduced. Following a similar approach to that first discussed in [12], we consider the formation of clusters of neurons, where neurons are connected only to the neurons in their cluster. As a result, with the exception of border-neurons, sensitivity is restricted to neurons in the same cluster. This implies that each hidden layer node i only communicates with N/B other nodes, where B is the number of clusters. The rest of the

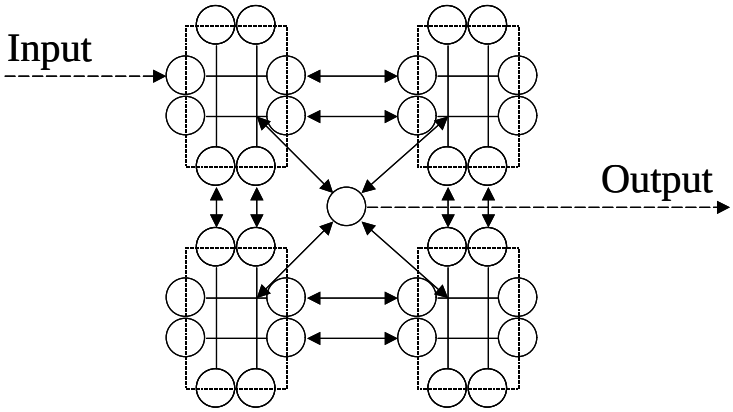

Figure 4-4: A diagram of 4 TRTRL clusters with 8 neurons in each. The shared neuron at the center is the output neuron.

architecture in this approach remains the same as before. In particular, all nodes receive input patterns from the input layer and all nodes have a link to the output layer. It should be noted that there must be some connectivity between clusters, defined as inter-cluster connectivity, thus some neurons within each cluster are connected to other neurons in other clusters by means of a direct link. A diagram of a cluster structure is illustrated in Figure 4-4.

4.2.1 Performance Comparison of Clustered and Nonclustered TRTRL

The benchmark used here is an electricity demand prediction task. The data set comprises of 15,240 data point, each of which is a 15 minutes averaged value of power demand in the full year 1997. The objective is to predict the power demand in 8 hours. The data is normalized before entering the neural networks. The clustered TRTRL consists of 4 clusters with 8 neurons in each cluster and a single shared output neuron. The nonclustered TRTRL has 25 neurons. The initial learning rate is set to 0.01 for both networks and initial weights are random.

Figure 4-5 shows the average results for 50 independent test. As one would expect, the clustered TRTRL learns slower initially, due to the reduced connectivity, but it eventually reaches the same accuracy as with nonclustered TRTRL. Note that more neurons are needed for clustered TRTRL to achieve the same performance as nonclustered TRTRL, however, the extra neurons are well justified because the algorithm is more localized and the computational

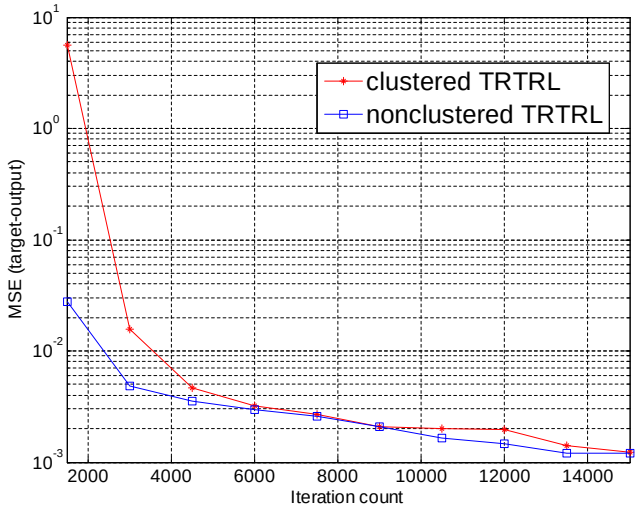

Figure 4-5: Comparison of clustered and nonclustered TRTRL

load is reduced.

4.3 Applying TRTRL RNNs in Solving POMDP

4.3.1 Direct-Policy Approximate DP with RTRL-RNN

In this section, we introduce `RNNPOMDP`, an online stochastic gradient learning control framework, which utilizes a recurrent neural network for Q-function approximation. The controller is constructed of a fully-connected RNN with one output neuron that predicts the state-action value, based on which the softmax algorithm [38] is used to determine the actions. The goal is to learn a stochastic control scheme that yields a near-optimal policy. All the RNN nodes use sigmoid activation function with the exception of the output node which has a linear activation function. We apply Q-learning and use the TRTRL-SMD algorithm to train the RNN. To that end, the RNN is trained with reference to the temporal difference error,

$$\delta_t = r_t + \gamma \max_i \{Q(\vartheta_t, a_i)\} - Q(\vartheta_{t-1}, a_{t-1}), \tag{4.27}$$

RNN $Q-$learning with softmax action selection
1. Given:
(a) an ergodic POMDP with observation space Θ, action space A, and bounded reward;
(b) an RNN with initial weights w_0 and function mapping observation-action pair to real value $f : \Theta \times A \rightarrow R$
2. For $t = 1$ to ∞ :
(a) Interact with POMDP:
1) observe $\vartheta_t \in \Theta$, evaluate all actions $a_i \in A(\vartheta_t)$ via the RNN via $Q(\vartheta_t, a_i) = f(\theta_t, a_i, w_t)$;
2) calculate the probability of taking each action in ϑ_t using softmax: $\Pr\{a_i \text{ in } \vartheta_t\} = \frac{e^{Q(\vartheta_t, a_i)/\tau}}{\sum_i e^{Q(\vartheta_t, a_i)/\tau}}$;
3) observe the reward r_{t+1};
(b) Update the activations (i.e. internal states) and weights of the RNN:
1) input the current observation-action pair (ϑ_t, a_t) to RNN;
2) update prior time step weights and sensitivities, based on $(\vartheta_{t-1}, a_{t-1})$,and (4.10), with temporal-difference error defined as $\delta_t = r_t + \max_i\{Q(\vartheta_t, a_i)\} - Q(\vartheta_{t-1}, a_{t-1})$;

Table 4.1: Q-function approximation based POMDP learning using the TRTRL-SMD algorithm.

where ϑ_t is the observation, a_i are within the set of possible actions, r_t denotes the single-step reward and γ is the discounting factor set to 0.8. For each step, the RNN is used to evaluate the Q-function for all possible actions, followed by softmax action selection. Since the neural network has state (by means of activations), previous activations and weights are stored and updated during the subsequent time step, upon evaluation of the temporal difference error. The algorithm is given in Table 4.1.

Simple Three-State POMDP

We first consider a simple 3-state POMDP, as used by Baxter et al. and Schraudolph et al. [38]. The RNN consisted of 5 internal neurons and one output neuron. The observation for each state is a vector like $(6/18, 12/18)$. Of the two possible transitions from each state, the preferred one occurs with 80% probability, the other with 20% as show in Fig (4-6). The preferred transition is determined by the action of a simple probabilistic adaptive controller that receives two state-

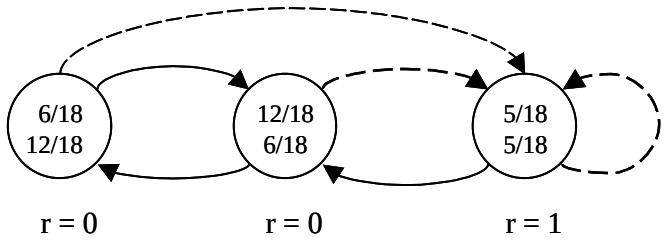

Figure 4-6: Baxter et al's simple 3-state POMDP. States are labelled with their observable feature vectors and instantaneous reward r; arrows indicate the 80% likely transition for the first (solid) *resp.* second (dashed) action.

dependent feature values as input, and is trained to maximize the expected average reward.

In this test, the free parameter for softmax algorithm was set to $\tau = 0.5$; for TRTRL-SMD, the parameters were configured to the following initial values: $\lambda_{ij}(0) = 0.01, \forall\ i, j$, $\rho = 0.5, \beta = 0.95, \mu_{ij}(0) = 0.1$, $\mu_{ij} \in [0.01, 0.5]$, and $\eta = 0.05$. We collected data from 500 independent runs with random seeds and initial conditions, and compared the convergence rates with the ones obtained for `SMDPOMDP` [38], which is a feed-forward neural network based approach. The comparison is shown in Fig 4-7. Both algorithms converge asymptotically to the optimal average reward ($R = 0.8$), with the `RNNPOMDP` algorithm converging faster in terms of process steps. This is consistent with the fact that RNNs have stronger approximation capabilities and faster convergence rate when compared to feed-forward networks.

Modified Three-State POMDP

The first test was a simple three-state POMDP with the property that greedy maximization of instantaneous reward leads to the optimal policy. Schraudolph et al. [38] introduced a more challenging problem which assigns to each state deceptive instantaneous reward. In the modified POMDP 4-8 , the highest reward state can only be reached through an intermediate state with a negative reward. The state features (observations) were also modified to create an ill-conditioned input to the controller. For this test, we used a 10-neuron RNN with the same setting as stated above, and train the network according to the algorithm described in table 4. Fig 4-9 illustrates that while `SMDPOMDP` reaches the optimal performance after approximately 10^6 iterations, `RNNPOMDP` converged to the optimal average reward almost 10

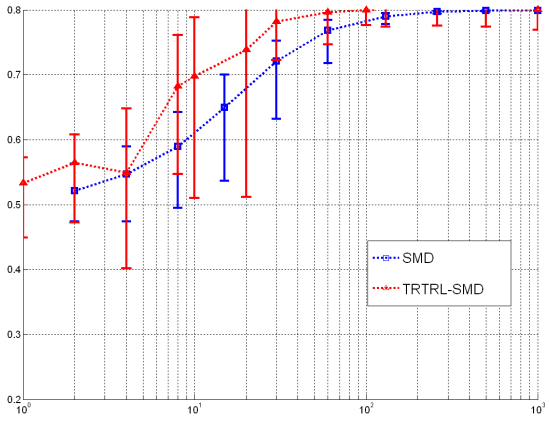

Figure 4-7: Comparison of regular SMD and TRTRL-SMD applied in simple 3-state POMDP

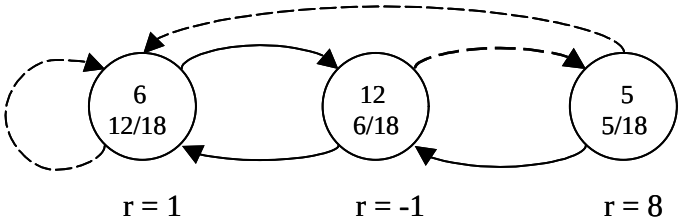

Figure 4-8: Schraudolph et al's modified 3-state POMDP

times faster. Moreover, the RNN trained with SMD also yielded considerable performance improvement prior to converging to the optimal average reward.. This further supports the notion that utilizing RNNs in this context results in better approximation of long-term rewards.

Four-State POMDP

While the translation of features to inferred states was indirect in the above test cases, it did not constitute a true POMDP in the sense that observation-to-state is ambiguous. We therefore studied a four-state POMDP, as depicted in Fig 4-10 . The two states on the left-hand side have the same observable features but different preferred actions. When the agent visits any of the two states on the right-hand side, it transitions to the states on the left-hand side (as depicted in the figure), regardless of the action taken. The observation is memory-dependent

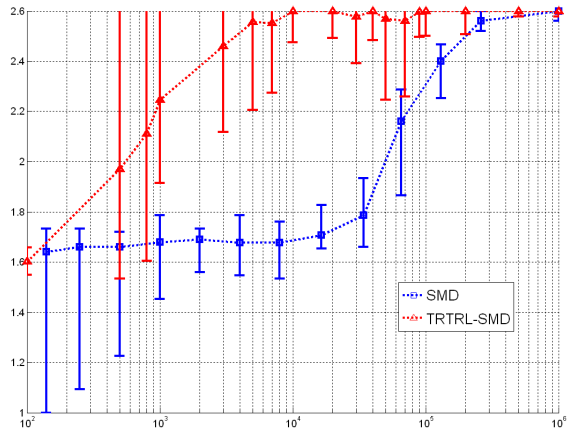

Figure 4-9: Comparison of regular SMD and RNN-SMD applied in modified 3-state POMDP

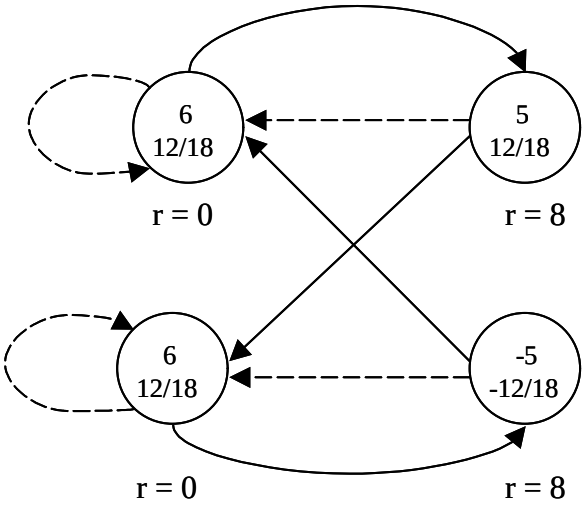

Figure 4-10: 4-state POMDP with identical observations for different states.

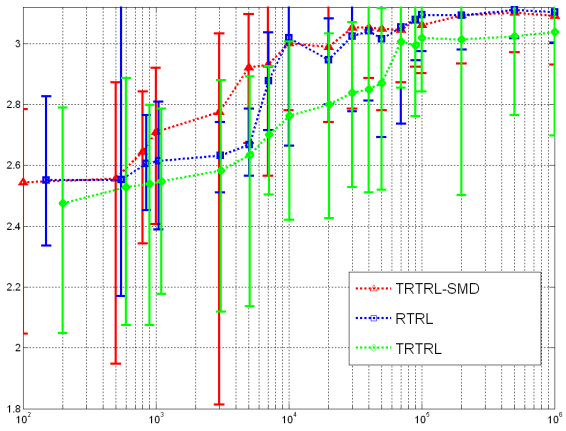

Figure 4-11: RNN-SMD applied in 4-state POMDP

in the sense that each state has a distinct preceding observations. The controller needs to memorize preceding observations in order to determine the optimal action for each of the two states. Hence, stateful function approximation, such that achieved by RNN, is mandatory. The RNN in this case consisted of 15-neurons with the same initial setup as stated above. Fig 4-11 demonstrates the asymptotic convergence to the policy. It is worth mentioning that a near-optimal policy is obtained after only 10^4 iterations.

Chapter 5

Consolidated Actor-Critic Model

Partially Observable Markov Decision Processes (POMDPs) characterize a broad range of real-world problems in which an agent interacts with its environment without being provided with an explicit state representation. In many practical scenarios identical observations may be provided for different states, thereby requiring the agent to rely on memory to infer its state. An agent in a path-searching problem (e.g. maze maneuvering) may receive identical observations for several different positions (or states). In such cases, the agent must recall recent steps in order to infer its precise position. Many problems of interest can be formulated as POMDPs, yet the lack of efficient algorithms results in the limited use of POMDPs in practice. In MDPs the agent's observation is equivalent to the environment's state. Therefore, the solution for MDPs is simply a mapping between observed states to actions. However, in a POMDP, such a memoryless or perception-based policy will not suffice, and thus the agent must construct an internal state-based policy.

5.1 Actor-Critic Models for Solving POMDPs

By using recurrent neural networks as value function approximators in the RL framework, an actor-critic learning architecture has been proposed in [39] to solve MDPs by means of Neural Dynamic Programming (NDP). The objective of this on-line learning control scheme is to optimize a desired performance measure by learning to choose appropriate control actions through interaction with the environment. This structure includes two networks, actor and

critic as fundamental building blocks. The critic uses an approximation architecture to learn a value function, which is then used to update the actor's policy parameters in the direction of performance improvement. In the case of solving POMDPs, the actor-critic model enables the agent to effectively address non-Markovian situations by relying on the agent's internal memory (state).

An agent with internal state can distinguish between different events in its past, and therefore can become sensitive to the non-Markovian dependencies that yield hidden states. When we focus on neural network (NN) function approximators, this requirement may be realized by networks with embedded context units, such as an Elman network or a Jordan network [14] [8]. These network use recurrent connections to infer state information as means of approximating the value function. To address this need, studies such as [15] investigated an actor-critic architecture where both actor and critic are fully recurrent neural networks, and both trained with RTRL. Moreover, in [40] another architecture is considered where the actor and critic modules share hidden and context layer neurons. While the weights for the critic network are updated using the Bellman error, updating the actor network is achieved via heuristic schemes based on the reinforcement signal generated from the critic network.

Here we propose a consolidated actor-critic model (CACM) for solving POMDPs in continuous state and continuous, multi-dimensional action spaces. The proposed design is biologically-inspired in that the consolidated architecture may be closer to how the mammal cortex is built, i.e. a single coherent module rather than two separate architectures, as in classical actor-critic methods. Also, observing that both the actor and critic perform some form of modeling the environment dynamics, consolidating them into one network efficiently reduces the resource requirements and computational complexity while retaining performance.

5.2 Related Work

In [40], actor-critic learning is implemented using a two-output single-hidden-layer Elman network with a subset of hidden nodes used as the context units. This architecture shares the hidden and context layers between "actor" and "critic" modules. Figure 5-1 depicts such an architecture, in which there are two different output neurons. One output neuron is the so-

called "critic" node, producing the current estimated total reward-to-go, which reflects on the expected quality of performance. The second neuron is the "actor" node, generating the current probability of choosing one of the actions, assuming that only two actions are allowed in each state.

The critic receives external (primary) reinforcement from the environment and transforms it into internal (heuristic) reinforcement for the actor. The weight update rule follows the usual error minimization scheme used in supervised learning. Specifically, the critic is updated using TD methods, while the actor is updated with the heuristic reinforcement signal translated by the critic. In other words, for a given action a, if the Bellman error is $r(t+1)+\gamma J(t+1) > J(t)$ (where γ is the discounting factor), this action is perceived as being good, and therefore that action should be reinforced by having the probability of choosing it increased. Conversely, if the reverse inequality holds, the action is undesired and thus should be inhibited.

This architecture is evolutionary in two aspects: first, it combines the actor network and critic network, which may be closer to biological neural networks in the mammal brain than the two separate network architecture as in classical actor-critic methods [40]. Second, in the ordinary Elman-network, all hidden activations are fed back to the context layer. As the number of hidden nodes increases, so does the number of context nodes. However, to capture historical features, it may not be necessary to use the entire hidden layer as the context layer; only some portion of the hidden layer can be used as the context layer, as illustrated in Figure 5-1.

However, the limitations of this method are: (1) it requires that an action must be a scalar taking only two possible values, rendering it unpractical in the context of many real world scenarios where the action set is much larger. Given that the output of the actor node is the probability of choosing an action $\Pr(a)$, it is implied that the probability of choosing another action is $1 - \Pr(a)$. Thus the output from the actor node would be confusing if more than two actions are presented for each state; (2) because the actor is updated heuristically using the internal reinforcement signal translated by the critic, the predictive error for the actor node is not accurate and optimal in the backpropagation process, leading to slow convergence rate.

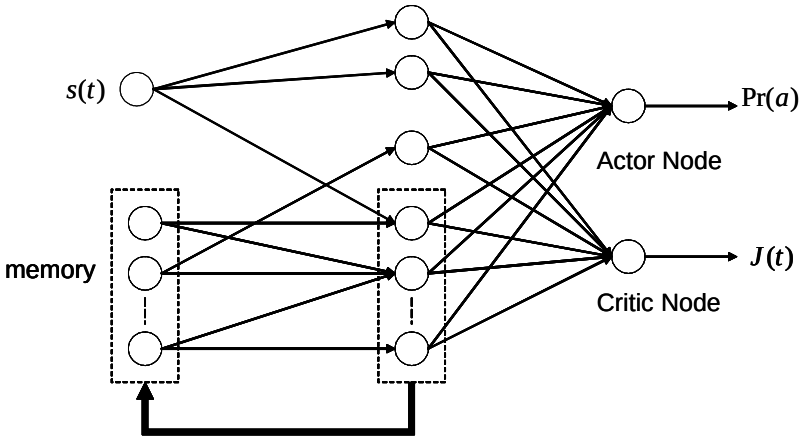

Figure 5-1: An actor-critic Elman network.

5.3 Motivation for the Consolidation of Actor and Critic Networks

To solve POMDPs, both the actor and critic networks must infer the actual system state from a sequence of observations, resulting in duplicated effort for the two networks. Hence, a primary motivation for the CACM approach is to consolidate the two networks into one in order to conserve the resources and computational load involved. As such, we extend the work of [40] in several ways: first, the action space for each state can be high-dimensional and continuous. Second, the embedded actor is updated in a strict backpropagation fashion, leading to faster convergence toward the optimal policy.

5.4 The Consolidated Actor-Critic Model (CACM)

The CACM design attempts to simplify the architecture and computations involved in classical actor-critic models, while retaining their convergence properties. At the core of the CACM is an RNN, the functionality of which is two fold: first, it generates the policy (the action for each individual state); second, it estimates the action value function (value of each state-action pair). The inputs to the CACM are the state feature vector $s(t)$ and the action $a(t)$ generated during the prior step (i.e. delayed by one time step); the outputs are the estimated cost-to-go

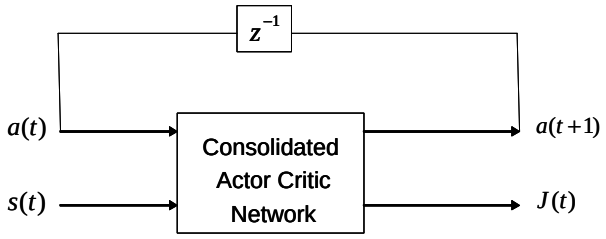

Figure 5-2: Consolidated Actor Critic Model

value $J(t)$ for the state-action pair $(s(t), a(t))$, and the action selected for the subsequent state, $a(t+1)$, as illustrated in 5-2.

It should be noted that $a(t+1)$ is fed back to the network for evaluation during the next time step. An obvious difference between the CACM and classical actor-critic approaches is that instead of generating the action $a(t+1)$ using the state $s(t+1)$, $a(t+1)$ is generated based on information regarding state and action at the proceeding step $(s(t), a(t))$. In this manner, CACM abbreviates the policy generation process: it first predicts the next state based on the current state and action, and then it maps the predicted state to an action.

5.5 CACM training with TRTRL

In this section, we study the application of TRTRL to the CACM framework employing a fully-connected recurrent neural network. The learning algorithm we use here is TRTRL with stochastic meta-descent, as described in the previous chapter. Figure 5-3 depicts the proposed network structure.

At time t, the state action pairs $(s(t), a(t)-\epsilon)$, $(s(t), a(t))$, $(s(t), a(t)+\epsilon)$ are evaluated and compared, where ϵ is a small positive value. The reinforcement signal is derived from finding the maximal of the three output values, $J_{-\epsilon}(t), J(t)$ and $J_{+\epsilon}(t)$. For example, if $J_{-\epsilon}(t) > J(t) > J_{+\epsilon}(t)$, a negative reinforcement signal is applied as update to the network. This leads to a simpler yet efficient updating scheme when incorporating TRTRL. The consolidated actor-critic network provides two outputs: $J(t)$ and $a(t+1)$. $J(t)$ is an approximation for $R(t)$, the weighted

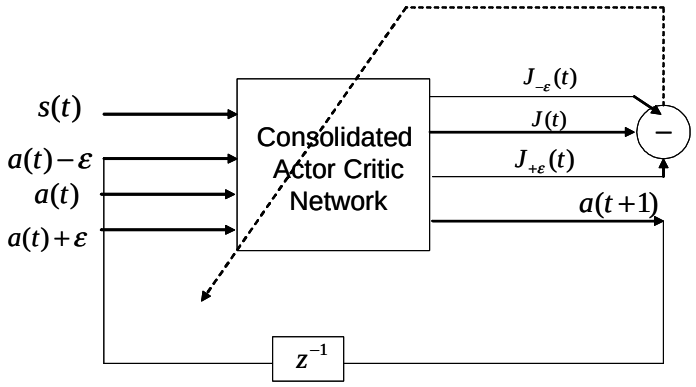

Figure 5-3: Consolidated Actor Critic with TRTRL

total future reward-to-go, which is given by

$$\begin{aligned} R(t) &= r(t+1) + \gamma r(t+2) + ... \\ &= \sum_{k=1}^{\infty} \gamma^{k-1} r(t+k), \end{aligned}$$

where $r(t+1)$ is the reward for the state action pair $(s(t), a(t))$ and γ is the discounting factor. $a(t+1)$ is the output control for the next state $s(t+1)$.

The prediction error (i.e. Bellman error) is defined as

$$e_c(t) = [r(t) + \alpha J(t)] - J(t-1),$$

and the objective function to be minimized is

$$E_c(t) = \frac{1}{2} e_c^2(t).$$

The action error is defined as

$$e_a(t) = J(t) - R^*.$$

Thus, we wish to minimize the following performance error measure:

$$E_a(t) = \frac{1}{2} e_a^2(t).$$

Let w denote the weight parameters in the CACM network. The CACM model can be represented by $[J, a] = nn(s, a, w)$, where nn denotes the consolidated actor-critic network. The gradient trace for the action error is given by $\frac{\partial E_a(t)}{\partial a(t)}$. The update algorithm for the consolidated network is a gradient-based adaptation given by

$$E(t) = E_c(t) + \frac{\partial E_a(t)}{\partial a(t)}$$

$$w(t+1) = w(t) + \Delta w(t),$$
$$\Delta w(t) = l(t)\left[-\frac{\partial E(t)}{\partial w(t)}\right],$$

where $l(t) > 0$ is the learning rate of the critic network at time t. It is noted here that the error for the CACM network consists of the error from the critic node and gradient of the error from the actor node.

5.5.1 The On-line Learning Algorithm

We next describe concrete implementation details of the CACM. A nonlinear multi-layer feed-forward network is the basic network used for the CACM investigated. In this design, a single hidden layer is utilized by the network. The general system diagram and relevant notations are provided in figure 5-4. As in the classic actor-critic method, the analysis has two components: a forward path which generates the estimated action values and selected next action, and a backward path that updates the parameters (weights) of the neural network. Consider a CACM with one input layer, one hidden layer and one output layer. The activation function of the neurons in input layer is linear, for the hidden layer is nonlinear 5.1 and for the first output (J value estimation) it is linear while for the second output (next action) it is linear too. Let the state, action and reward all be scalars. The network is illustrated in figure 5-4. We next define the input to the consolidated network as $u(t) = [a(t),\ s(t),\ y(t-1)]$, where $y(t-1)$ is the output of the hidden layer during the previous time step. In the CACM network, the output $J(t)$ is of the form

$$J(t) = \sum_{i=1}^{N_h} w_{ci}^{(2)}(t) y_i(t),$$

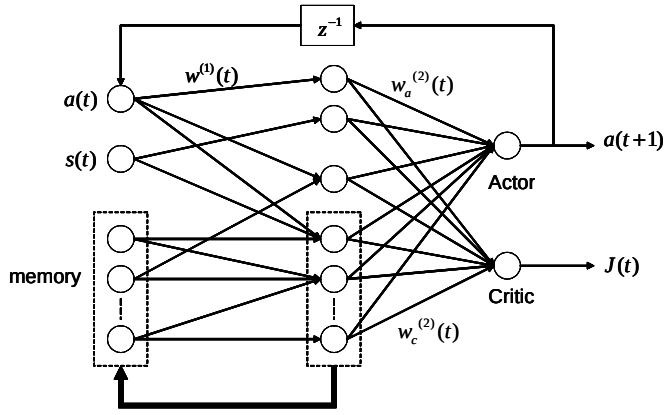

Figure 5-4: Neural Network Implementation of Consolidated Actor Critic Model

where N_h is the number of neurons in the hidden layer, $y_i(t)$ is the output of the i^{th} node in the hidden layer and w_{ci} denotes the weights between the i^{th} node in the hidden layer and the critic output node. Correspondingly, the action output, $a(t+1)$, is of the form

$$a(t+1) = \sum_{i=1}^{N_h} w_{ai}^{(2)}(t) y_i(t),$$

where $w_{ai}^{(2)}$ represents the weights between the i^{th} node in the hidden layer and the actor output node. Further, we compute $y_i(t)$ as

$$y_i(t) = \frac{1 - e^{-x_i(t)}}{1 + e^{-x_i(t)}}, \quad i = 1, ..., N_h, \tag{5.1}$$

$$x_i(t) = \sum_{j=1}^{n} w_{ij}^{(1)}(t) u(t), i = 1, ..., N_h$$

where n is the dimension of the input vector and $w_{ij}^{(1)}(t)$ is the weights between the input and hidden layers. Memory units are represented by $y_j(t-1)$ in the expression above. By applying the chain rule, we first backpropagate the action error,

$$\frac{\partial E_a(t)}{\partial a(t)} = \sum_{i=1}^{N_h} \frac{\partial E_a(t)}{\partial y_i(t)} \frac{\partial y_i(t)}{\partial x_i(t)} \frac{\partial x_i(t)}{\partial a(t)}$$

The weight parameters are updated according to the following steps:

- $\Delta w_a^{(2)}$ (hidden to actor node):

$$\Delta w_{ia}^{(2)}(t) = l(t)\left[-\frac{\partial E_a(t)}{\partial a(t)}\frac{\partial a(t)}{\partial w_{ia}^{(2)}(t)}\right],$$

$$\begin{aligned}\frac{\partial E_a(t)}{\partial a(t)}\frac{\partial a(t)}{\partial w_{ia}^{(2)}(t)} &= \sum_{i=1}^{N_h}\frac{\partial E_a(t)}{\partial y_i(t)}\frac{\partial y_i(t)}{\partial x_i(t)}\frac{\partial x_i(t)}{\partial a(t)}\frac{\partial a(t)}{\partial w_{ia}^{(2)}(t)} \\ &= \sum_{i=1}^{N_h}\frac{\partial E_a(t)}{\partial J(t)}\frac{\partial J(t)}{\partial y_i(t)}\frac{\partial y_i(t)}{\partial x_i(t)}\frac{\partial x_i(t)}{\partial a(t)}\frac{\partial a(t)}{\partial w_{ia}^{(2)}(t)} \\ &= e_a(t)\left(\sum_{i=1}^{N_h} w_{ci}^{(2)}(t)\left[\frac{1}{2}\left(1-y_i^2(t)\right)\right] w_{ia}^{(1)}(t) y_i(t)\right),\end{aligned}$$

  where $w_{ia}^{(1)}(t)$ is the weight between the action and i^{th} node in the hidden layer.

- $\Delta w_c^{(2)}$ (hidden to critic node):

$$\Delta w_{ic}^{(2)}(t) = l(t)\left[-\frac{\partial E_c(t)}{\partial w_{ic}^{(2)}(t)}\right],$$

$$\frac{\partial E_c(t)}{\partial w_{ic}^{(2)}(t)} = \frac{\partial E_c(t)}{\partial J(t)}\frac{\partial J(t)}{\partial w_{ic}^{(2)}(t)} = e_c(t) y_i(t).$$

- $\Delta w^{(1)}$ (input to hidden layer):

$$\Delta w_{ij}^{(1)}(t) = l(t)\left[-\frac{\partial E(t)}{\partial w_{ij}^{(1)}(t)}\right],$$

$$\begin{aligned}\frac{\partial E(t)}{\partial w_{ij}^{(1)}(t)} &= \frac{\partial E_c(t)}{\partial w_{ij}^{(1)}(t)} + \frac{\partial E_a(t)}{\partial a(t)}\frac{\partial a(t)}{\partial w_{ij}^{(1)}(t)} \\ &= \left(\frac{\partial E_c(t)}{\partial J(t)}\frac{\partial J(t)}{\partial y_i(t)} + \frac{\partial E_a(t)}{\partial a(t)}\frac{\partial a(t)}{\partial y_i(t)}\right)\frac{\partial y_i(t)}{\partial x_i(t)}\frac{\partial x_i(t)}{\partial w_{ij}^{(1)}(t)}\end{aligned}$$

Pseudocode implementing CACM method
Initialize w arbitrarily Repeat (for each trial) Initialize network input as $u(t) = [s(t) \;\; y(t-1)]$ Repeat (for each step t of the trial): Feedback the action from previous step $a(t)$ Calculate estimated reward-to-go: $J(t) = nn\,(u(t), a(t))$ Choose the action for the next state: $a(t+1) = nn\,(u(t), a(t))$ Repeat (updating w): $e_c(t) = [r(t) + \alpha J(t)] - J(t-1)$ $E_c(t) = \frac{1}{2} e_c^2(t)$ $e_a(t) = J(t) - R^*$ $E_a(t) = \frac{1}{2} e_a^2(t)$. $E(t) = E_c(t) + \frac{\partial E_a(t)}{\partial a(t)}$ $\Delta w(t) = l(t) \left[-\frac{\partial E(t)}{\partial w(t)} \right]$ $w(t+1) = w(t) + \Delta w(t)$ until maximum iteration number is reached $u(t) \longleftarrow u(t+1)$; $J(t-1) \longleftarrow J(t)$ until $s(t)$ is terminal until maximal trial is reached

Table 5.1: Pseudocode implementing CACM method.

$$= \left[e_c(t) w_{ic}^{(2)}(t) + e_a(t) \left(\sum_{i=1}^{N_h} w_{ci}^{(2)}(t) \left[\frac{1}{2} \left(1 - y_i^2(t) \right) \right] w_{ia}^{(1)}(t) \right) w_{ia}^{(2)}(t) \right] \times \left[\frac{1}{2} \left(1 - y_i^2(t) \right) \right] u_j(t)$$

Pseudocode for implementing the CACM method is summarized in the table 5.1.

5.6 Performance Evaluation

5.6.1 Cart-pole Balancing

The classic cart-pole balancing example is used to evaluate the CACM. This problem is often used as an example of inherently unstable and dynamic systems to demonstrate both modern and classic control techniques or reinforcement learning schemes, and is used here as a control benchmark. As depicted in figure 5-5, the cart–pole balancing problem is the problem of learning how to balance an upright pole. The bottom of the pole is hinged to the left or right of a cart

Parameters used in cart-pole system	
g	$-9.8\ m/s^2$
m_c	$1.0\ kg$
m	$0.1\ kg$
l	$0.5\ m$
F	force applied to the cart's center in newtons

Table 5.2: Parameters used in cart-pole system.

that travels along a finite-length track. The cart can only move in the horizontal direction, while the pole can only rotate about the point at which it is attached; that is, each has only one degree of freedom.

There are four state variables in the system: θ, the angle of the pole in an upright position (in degrees); x, the horizontal position of the cart's center (in meters); $\dot{x}$, the velocity of the cart (in m/s); $\dot{\theta}$, angular velocity (in degree/s). The only control action is f, which is the amount of force (in N) applied to the cart to move it left or right; the range of f in this experiment is $[-1, 1]$. The system fails when the pole falls past a certain angle (12 is used here) or when the cart runs into the boundary of the track (the distance is 2.4 m from the center to each boundary of the track). The system is modeled by the following nonlinear differential equations:

$$\ddot{\theta}_t = \frac{g \sin\theta_t + \cos\theta_t \left[\frac{-F_t - ml\dot{\theta}_t^2 \sin\theta_t + \mu_c sgn(\dot{x}_t)}{m_c + m} \right] - \frac{\mu_p \dot{\theta}_t}{ml}}{l \left[\frac{4}{3} - \frac{m\cos^2\theta_t}{m_c + m} \right]}$$

$$\ddot{x}_t = \frac{F_t + ml \left[\dot{\theta}_l^2 \sin\theta_t - \ddot{\theta}_t \cos\theta_t \right] - \mu_c sgn\left(\dot{x}_t\right)}{m_c + m}$$

The parameters used in this experiment are summarized in Table 5.2.

The goal of the controller is to determine a sequence of forces that, when applied to the cart, balance the pole so that it is kept upright. A control strategy was deemed successful if it balanced a pole for 100 000 time steps.

We use the cart-pole balance testbench to evaluate our CACM with fully connected recurrent neural network using both the TRTRL-SMD learning algorithm and a classic Elman network. We compare the performance of the CACM with a standard a actor-critic scheme. For the CACM/Elman implementation, a network consisting of one hidden layer with 24 nodes, 16 of

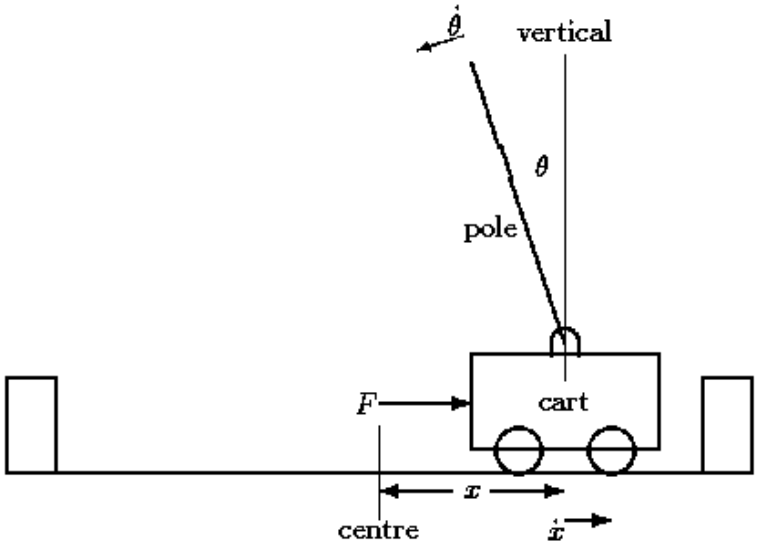

Figure 5-5: The cart-pole balancing system used [1].

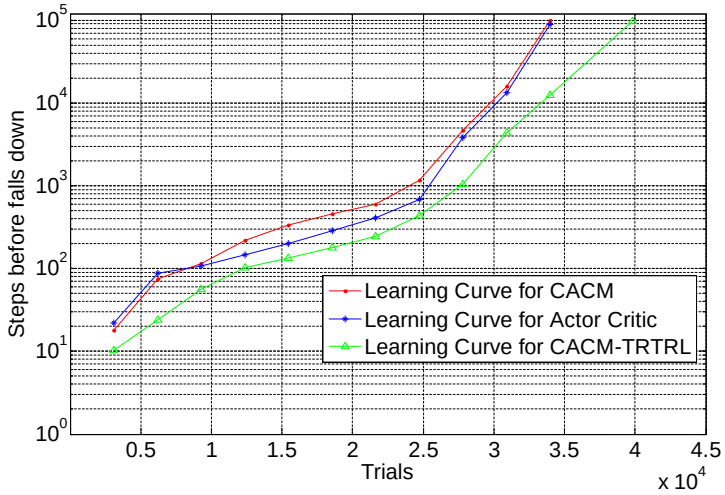

Figure 5-6: Comparison of learning performance between CACM (with Elman and with TRTRL-SMD) and the classical actor-critic method.

which are fed back to the input as context units. For the classical actor-critic architecture, the actor network is an Elman network consisting of one hidden layer with 16 neurons and the critic network is also an Elman network with one hidden layer having 24 neurons. Figure 5-6 depicts the learning process, averaged over 20 runs, for both the CACM and actor-critic methods. The $x-$axis denotes the number of trials before a successful balancing and $y-$axis the number of steps in each trial before the pole falls down. The results clearly demonstrates that by consolidating the actor and critic network, CACM achieves at least the same if not better performance when compared to a separate actor-critic system. This could be easily explained by the fact that redundant model learning has been avoided, thus improving the learning rate. Moreover, we eliminated the neurons and weights from the actor network thus effectively conserving resources and computations.

Chapter 6

Summary of Contributions

This dissertation has focused on pragmatic approaches to effectively scaling reinforcement learning based systems, in both tabular and function-approximation based frameworks. For the latter, particular attention has been given to partially-observable problems, which characterize many practical problems. Moreover, the different components contribute to the overall effort of facilitating hardware realization in next-generation RL systems. The following outlines the key contributions made.

6.1 Convergence Proof of Q-Learning with Delays

In chapter 3, we have shown that Q-Learning can be applied to MDPs with observation, action and cost delays. MDPs with delays can be treated as a partially observable MDPs, where the agent has no instantaneous system state information and the actions it issues experience delay, as is the case in many pragmatic applications. We have identified optimal values for such scenarios and proved respective convergence properties. The assertions made were based on previous work which has shown that the optimal policy for an MDP with delays can be computed by utilizing an equivalent MDP without delays. We have extend this prior work by proving convergence of Q-Learning variants that pertain to MDPs for which the costs and transition probabilities are unknown.

6.2 Truncated Real Time Recurrent Learning with Stochastic Meta-Descent

In chapter 4, we presented TRTRL-SMD - a framework for substantially reducing the resource requirements of learning in recurrent neural network, while retaining high-performance. The method is based on limiting the sensitivities of neuron activations to weights associated with either incoming or outgoing links, coupled with employing SMD - an efficient stochastic gradient descent method. Based on standard testbench cases, it is demonstrated through simulations that the performance of TRTRL-SMD exceeds that of RTRL, while speed and storage requirements are significantly reduced. Moreover, the clustered TRTRL is proposed to further reduce the computation and storage while maintain the essential properties of RTRL. The comparison graph shows that non-clustered TRTRL outperforms clustered TRTRL by a small number, which can be eliminated by increasing the number of clusters or the number of neurons per cluster in clustered TRTRL.

6.3 NeuroDynamic Programming with TRTRL

In chapter 4, we also presented a recurrent neural network based Q-learning POMDP framework. An efficient realization of the RNN yielded a scalable architecture, while training was improved via the stochastic-meta descent technique. Simulation results applied to several POMDP test cases clearly demonstrated the performance advantages of the proposed scheme, with respect to both accuracy in estimating the average reward as well as convergence properties. As part of this effort, we have improved the core SMD technique by further adapting learning parameters as a function of the data processed.

6.4 The Consolidated Actor-Critic Model

In chapter 5, we presented a novel recurrent neural network based consolidated actor-critic model for solving complex POMDPs. The architecture recognizes the fact that the actor and critic model both need to model the environment dynamics and thus propose a compact network that overcomes the inefficiency of standard actor-critic systems, yielding a more resource-

efficient and lower-complexity solution. The cart-pole balance simulation clearly demonstrated the performance and effectiveness of the proposed scheme, with regards to both convergence accuracy and speed.

6.5 Relevant Publications

The following is a list of publications pertaining to contributions made thus far, as described in this proposal:

- **Zhenzhen Liu** and Itamar Elhannay, "A Consolidated Actor Critic Model for Solving Large-Scale POMDPs," in preparation for submission.
- **Zhenzhen Liu** and Itamar Elhanany, "High-Speed Q-Learning Hardware Architecture for Large Action Sets," *IEEE Midwest Symposium on Circuits and Systems (MWSCAS), 2007.*
- **Zhenzhen Liu** and Itamar Elhanany, A Fast and Scalable Recurrent Neural Network based on Stochastic Meta-Descent," to appear in *IEEE Transactions on Neural Networks.*
- **Zhenzhen Liu** and Itamar Elhanany, "Fast and Scalable Recurrent Neural Network Learning based on Stochastic Meta-Descent," the *26th American Control Conference (ACC),* New York City, July 11-13, 2007.
- **Zhenzhen Liu** and Itamar Elhanany, "A Scalable Model-Free Recurrent Neural Network Framework for Solving POMDPs," *2007 IEEE International Symposium on Approximate Dynamic Programming and Reinforcement Learning (ADPRL),* April 1-5, 2007.
- **Zhenzhen Liu** and Itamar Elhanany, "RL-MAC: A Reinforcement Learning based MAC Protocol for Wireless Sensor Networks," the *International Journal of Sensor Networks,* Vol.1, No.2, 2006.
- **Zhenzhen Liu** and Itamar Elhanany, "RL-MAC: A QoS-Aware Reinforcement Learning based MAC Protocol for Wireless Sensor Networks," *IEEE Conference on Networking, Sensing and Control,* Ft. Lauderdale, FL, April 23-25, 2006.

Bibliography

Bibliography

[1] S. Grant, "Modelling cognitive aspects of complex control tasks," *Proceedings of the IFIP TC13 Third Interational Conference on Human-Computer Interaction*, pp. 1017–1018, 1990.

[2] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Cambridge MA, MIT Press, 1998.

[3] K. Murphy, "A survey of pomdp solution techniques," *K. Murphy. A survey of POMDP solution techniques. Technical Report, U.C. Berkeley, 2000.*, 2000. [Online]. Available: citeseer.ist.psu.edu/murphy00survey.html

[4] D. P. Bertsekas, *Dynamic programming and optimal control*, 3rd ed., 2007, vol. 2.

[5] C. J. Watkins, "Learning from delayed rewards," *PhD thesis*, 1989.

[6] M. Jordan, "Serial order: A parallel distributed processing approach," Institute for Cognitive Science Report 8694. University of California, San Diego, 1986.

[7] J. L. Elman, "Finding structure in time," *Cognitive Science*, no. 14, pp. 179–211, 1990.

[8] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning internal representation by error propagation," in *Parallel Distributed Processing: Explorations in the Microstructure of Condition*, D. E. Rumelhart and J. L. McClelland, Eds. Cambridge, MA: MIT Press, Bradford Books, 1986, vol. 1, pp. 318–362.

[9] P. Werbos, "Backpropagation through time: what it does and how to do it," *Special issue on neural networks, Proceedings of IEEE*, vol. 78, pp. 1550–1560, October 1990.

[10] R. J. Williams and J. Peng, "An efficient gradient-based algorithm for on-line training of recurrent network trajectories," *Neural Computation*, vol. 2, pp. 490–501, 1990.

[11] B. A. Pearlmutter, "Learning state space trajectories in recurrent neural networks," *Neural Computation*, vol. 1, pp. 263–269, 1989.

[12] D. Zipser, "A subgrouping strategy that reduces complexity and speeds up learning in recurrent networks," *Neural Computation*, no. 1, pp. 552–558, 1989.

[13] J. Si, A. G. Barto, W. B. Powell, and D. W. II, *Handbook of learning and approximate dynamic programming*. Wiley-IEEE Press, August 2004.

[14] P. Werbos, "Approximate dynamic programming for real-time control and neural modeling," *Handbook of Intelligent Control*, 1992.

[15] J. H. Schmidhuber, "Networks adjusting networks," 1990. [Online]. Available: citeseer.ist.psu.edu/schmidhuber90network.html

[16] L. Meeden, G. McGraw, and D. Blank, "Emergent control and planning in an autonomous vehicle," pp. 735–740, 1993. [Online]. Available: citeseer.ist.psu.edu/meeden93emergent.html

[17] L.-J. Lin and T. M. Mitchell, "Reinforcement learning with hidder states," *Proceedings of the second international conference on From animals to animats 2 : simulation of adaptive behavior: simulation of adaptive behavior*, pp. 271 – 280, 1993.

[18] K. Katsikopoulos and S. Engelbrecht, "Markov decision processes with delays and asynchronous cost collection," *IEEE Transactions on Automatic Control*, vol. 48, no. 4, pp. 568–574, 2003.

[19] E. Altman and P. Nain, "Closed-loop control with delayed information," *Perf. Eval. Rev.*, vol. 14, pp. 193–204, 1992.

[20] J. L. Bander and C. C. W. III, "Markov decision processes with noise-corrupted and delayed state observations," *J. Opl. Res. Soc.*, vol. 50, pp. 660–668, 1999.

[21] P. Varaiya and J. Walrand, "On delayed sharing patterns," *IEEE Transactions on Automatic Control*, vol. 23, no. 3, pp. 443–445, 1978.

[22] K. Hsi and S. I. Marcus, "Decentralized control of finite state markov processes," *IEEE Transactions on Automatic Control*, vol. AC-27, pp. 426–431, 1982.

[23] D. P. Bertsekas and J. N. Tsitsiklis, *Neuro-Dynamic Programming.* Belmont, MA: Athena Scientific, 1996.

[24] C. J. C. H. Watkins and P. Dayan, "Q-learning," *Mach. Learn.*, vol. 8, pp. 279–292, 1992.

[25] 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, IEEE Transactions on*, vol. 36, no. 1, pp. 92–106, 2006.

[26] M. J. Er and C. Deng, "Online tuning of fuzzy inference systems using dynamic fuzzy q-learning," *Systems, Man and Cybernetics, Part B, IEEE Transactions on*, vol. 34, no. 3, pp. 1478–1489, 2004.

[27] K.-S. Hwang, S.-W. Tan, and C.-C. Chen, "Cooperative strategy based on adaptive q-learning for robot soccer systems," *IEEE Transactions on Fuzzy Systems*, vol. 12, no. 4, pp. 569–576, 2004.

[28] J. Tsitsiklis, "Asynchronous stochastic approximation and q-learning," *Mach. Learn.*, vol. 16, pp. 185–202, 1994.

[29] N. N. Schraudolph, "Local gain adaptation in stochastic gradient descent," *Tech. Rep. IDSIA-09-99*, Aug 1999.

[30] J. Kivinen and M. Warmuth, "Additive versus exponentiated gradient updates for linear prediction," *Proc. 27th Annual ACM Symposium on Theory of Computing*, pp. 209–218, May 1995, new York, NY.

[31] S. E. Fahlman, "An empirical study of learning speed in back-propagation networks," *Computer Science Technical Report*, 1988.

[32] N. N. Schraudolph, J. Yu, and D. Aberdeen, "Fast online policy gradient learning with SMD gain vector adaptation," *19th Annual Conference on Neural Information Processing Systems*, Dec 2005, vancouver, Canada.

[33] B. A. Pearlmutter, "Fast exact multiplication by the Hessian," *Neural Computation*, vol. 6, no. 1, pp. 147–160, 1994.

[34] G. D. Magoulas, M. N. Vrahatis, and G. S. Androulakis, "Improving the convergence of the backpropagation algorithm using learning rate adaptation methods," *Neural Computation*, vol. 11, no. 7, pp. 1769–1796, 1999.

[35] T. tollenaere, "Supersab: fast adaptive backpropagation with good scaling properties," *Neural Networks*, vol. 3, no. 5, pp. 561–573, 1990.

[36] M. Mackey and L. Glass, "Oscillation and chaos in physiological control systems," *Science*, vol. 197, pp. 287–289, July 1977.

[37] R. C. III, "Predicting the mackey-glass timeseries with cascade-correlation learning," in *D. Touretzky, G. Hinton and T. Sejnowski eds., Connectionist Models Summer School Proceedings*, 1990, pp. 117–123, carnegie Mellon University.

[38] N. Schraudolph, J. Yu, and D. Aberdeen, "Fast online policy gradient learning with smd gain vector adaptation," in *Advances in Neural Information Processing Systems 18*, Y. Weiss, B. Schölkopf, and J. Platt, Eds. Cambridge, MA: MIT Press, 2006, pp. 1185–1192.

[39] W. B. P. Jennie Si, Andrew G. Barto and D. W. II, *Handbook of Learning and Approximate Dynamic Programming*. Wiley Interscience, 2004.

[40] E. Mizutani and S. E. Dreyfus, "Totally model-free reinforcement learning by actor-critic elman networks in non-markovian domains." [Online]. Available: citeseer.ist.psu.edu/325130.html

Vita

Zhenzhen Liu was born in Taihe, Jiangxi, People's Republic of China, on April 2nd, 1984. After finishing high school in 2000, she attended Nanjing University of Posts and Telecommunications, Nanjing, P.R.China, where she received a Bachelor of Engineering degree in 2004. Between August 2004 and December 2004, she worked as a software engineer in Amoi Electronic co., Ltd., Xiamen, P.R.China. In 2005, she came to study at the University of Tennessee, Knoxville, United States. She received a Doctor of Philosophy degree in computer engineering in Fall 2007, from the department of Electrical Engineering and Computer Science at the University of Tennessee, Knoxville.