2016

Intrinsically Motivated Exploration in Hierarchical Reinforcement Learning

Christopher M. Vigorito
University of Massachusetts - Amherst

INTRINSICALLY MOTIVATED EXPLORATION IN HIERARCHICAL REINFORCEMENT LEARNING

A Dissertation Presented

by

CHRISTOPHER M. VIGORITO

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

February 2016

College of Information and Computer Sciences

INTRINSICALLY MOTIVATED EXPLORATION IN HIERARCHICAL REINFORCEMENT LEARNING

A Dissertation Presented

by

CHRISTOPHER M. VIGORITO

To Mom, Dad, Diana, and Mena, for their unconditional support.

ACKNOWLEDGMENTS

First and foremost I am eternally grateful to Andy Barto for his stellar advisorship during my graduate career. His breadth of knowledge in the field and uncanny talent for providing clear and simple explanations for seemingly impenetrable topics have allowed me to escape many a technical quagmire and progress steadily in my research. Our interactions have provoked countless interesting discussions on several research topics, only a fraction of which I can realistically hope to pursue in a single thesis. His balance of constructive criticism and generous praise have greatly improved my skills as a scientist and technical writer, and brought me through the inevitable periods of frustration and doubt that accompany any significant research endeavor.

I would also like to thank the other members of my committee, Rod Grupen, Sridhar Mahadevan, and Neil Berthier for their invaluable feedback and expertise. I am fortunate to have pursued my degree at an institution where I could have direct and personal conversations with such leaders in their respective fields. No successful graduate career is possible without the aid of colleagues with whom to both commiserate and share friendly discussions, technical or otherwise. I am thusly indebted to the members of the Autonomous Learning Lab, past and present, especially George Konidaris, Scott Kuindersma, Sarah Osentoski, Ashvin Shah, Andrew Stout, Philip Thomas, Scott Niekum, Kimberly Ferguson, Will Dabney, Jeff Johns, Bruno Castro da Silva, Colin Barringer, Anders Jonsson, Ozgur Simsek, Pippin Wolfe, and Chang Wang.

I am grateful as well to my more recent colleagues, the team at Osaro, Inc., for their flexibility in my time off of work during the final month of completing my dissertation. I'm ecstatic to have joined such a talented and friendly team, and I'm looking forward to tackling the challenges we face in scaling reinforcement learning to real world problems. Special thanks also to Xander Dunn at Osaro for assisting in the implementation of the simulations for Chapter 3. Finally, I would like to thank my family, especially my parents, Michael and Carolyn, my sister Diana, and Mena for their unwavering and unconditional support throughout my academic career.

ABSTRACT

INTRINSICALLY MOTIVATED EXPLORATION IN HIERARCHICAL REINFORCEMENT LEARNING

FEBRUARY 2016

CHRISTOPHER M. VIGORITO

B.Sc., AMHERST COLLEGE

M.Sc., UNIVERSITY OF MASSACHUSETTS AMHERST

Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Andrew G. Barto

The acquisition of hierarchies of reusable skills is one of the distinguishing characteristics of human intelligence, and the learning of such hierarchies is an important open problem in computational reinforcement learning (RL). In humans, these skills are learned during a substantial developmental period in which individuals are *intrinsically motivated* to explore their environment and learn about the effects of their actions. The skills learned during this period of exploration are then reused to great effect later in life to solve many unfamiliar problems very quickly. This thesis presents novel methods for achieving such developmental acquisition of skill hierarchies in artificial agents by rewarding them for using their current skill set to better understand the effects of their actions on unfamiliar parts of their environment, which in turn leads to the formation of new skills and further exploration, in a life-long process of hierarchical exploration and skill learning.

In particular, we present algorithms for intrinsically motivated hierarchical exploration of Markov Decision Processes (MDPs) and finite factored MDPs (FMDPs). These methods integrate existing research on temporal abstraction in MDPs, intrinsically motivated RL, hierarchical decomposition of finite FMDPs, Bayesian network structure learning, and information theory to achieve long-term, incremental acquisition of skill hierarchies in these environments. Moreover, we show that the skill hierarchies learned in this fashion afford an agent the ability to solve novel tasks in its environment much more quickly than solving them from scratch.

To apply these techniques to environments with representational properties that differ from traditional MDPs and finite FMDPs requires methods for incrementally learning transition models of environments with such representations. Taking a step in this direction, we also present novel methods for incremental model learning in two other types of environments. The first is an algorithm for online, incremental structure learning of transition functions for FMDPs with continuous-valued state and action variables. The second is an algorithm for learning the parameters of a predictive state representation, which serves as a model of partially observable dynamical systems with continuous-valued observations and actions. These techniques serve as a prerequisite to future work applying intrinsically motivated skill learning to these types of environments.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 Motivation

A critical source of the versatility and robustness of human behavior is a substantial period of childhood development characterized by extensive exploration. During this period, useful abstractions are extracted from the statistical properties of one's environment and specialized skills are learned that reliably manipulate specific features present in those abstractions. Furthermore, the vast majority of the exploration conducted during this period is not motivated by immediate biological necessity. Rather, humans and other mammals seem *intrinsically* motivated to explore and manipulate their environment—that is, they engage in this behavior for its own sake, even though it does not confer any immediate survival advantages (Harlow et al., 1950). One explanation for the evolution of such behavior is the acquisition of skills it engenders, abstract behaviors that become useful later in an one's life when faced with challenging problems that *are* directly connected to survival and that would otherwise be too difficult to solve without such an existing skill set. The research presented in this dissertation takes inspiration from these biological phenomena and presents methods for the design of artificial reinforcement learning (RL) agents whose environmental structure affords developmental skill-learning curricula similar to those of biological agents.

Although much of the computational reinforcement learning literature is devoted to methods that efficiently search for solutions to distinct sequential decision problems (Sutton & Barto, 1998), in this work we focus rather on the scenario in which an agent may face many distinct tasks over the course of its lifetime, all of which share some common structure, and some of which must be solved before learning solutions to others can be attempted successfully. It has been argued that in such a scenario an agent that is intrinsically motivated to learn a hierarchical set of abstract skills will possess superior problem-solving abilities in complex tasks presented to it at a later period, as compared with an agent who tries to solve each of those later problems in isolation (Barto et al., 2004). In both the psychology and machine learning communities, this savings based on prior experience is known as *transfer* (Brown & Kane, 1988; Pan & Yang, 2010). We present here a framework in which such a hierarchy may be incrementally and autonomously learned via intrinsic motivation, and which allows for successful transfer of learned skills to the efficient solution of novel tasks. During the period in which these skills are acquired, there may be no explicit task the agent is required to perform, and the agent is thus free (and intrinsically motivated) to explore its environment much like an infant, extracting useful statistical relationships from its surroundings that enable abstract skill learning. The skills learned during this exploratory period are then put to use later in life, when the agent is faced with more complex *extrinsic* objectives.

In order to mimic natural, self-motivated learning and the environments in which it is generally observed, the skills acquired along this developmental pathway should be learned in an incremental, bootstrapped manner, increasing in complexity over time. An agent's current skill set and knowledge of its environment's dynamics at any given time provide a substrate for learning skills of a certain level of complexity. As skills of that level are mastered and added to the agent's behavioral repertoire, along with related knowledge about their effects on the environment, the augmented substrate provides the opportunity for learning skills and knowledge of still greater complexity. This process continues throughout the agent's life, always bootstrapped by its current expertise in manipulating its surroundings and its ability to reason about such manipulations.

In the remainder of this thesis, we describe a framework in which artificial reinforcement learning agents are intrinsically motivated to improve the accuracy of their environmental models, and to use those models in service of learning skills that reliably manipulate their environment. We present methods for achieving this behavior in environments represented formally as traditional Markov Decision Processes (MDPs) and finite factored MDPs (FMDPs). The results of employing these methods show that the hierarchical set of abstract skills such agents are motivated to learn provide them with the expertise necessary to efficiently solve novel problems with which they have never before been confronted. These agents make use of their acquired skills to transfer procedural knowledge gained while experimenting with their surroundings early in life to the solution of complex tasks later in life that would be too difficult to solve feasibly without such a skill set.

The extension of these model-based methods to environments with representational properties that differ from traditional MDPs and finite FMDPs requires methods for learning from experience formal transition models of environments with such representations. We thus also present in this work novel methods for model learning in two kinds of environments. The first is an algorithm for online, incremental structure learning of transition functions for FMDPs with continuous-valued state and action variables. The second is an algorithm for learning the parameters of a predictive state representation that serves as a model of a partially observable dynamical system with continuous-valued observations and actions. We show how these techniques serve as a prerequisite to future work pertaining to the application of intrinsically motivated skill learning in these kinds of environments.

1.2 Contributions

The work presented in this thesis contains four novel contributions to the machine learning community. These are summarized in the following sections, each expounded in greater detail in subsequent chapters.

1.2.1 Intrinsically Motivated Skill Learning in Markov Decision Processes

We present a framework for long-term, incremental learning of abstract skill hierarchies useful over ensembles of related tasks in environments formalized as Markov Decision Processes (MDPs). An agent in this framework learns skills incrementally as enough structural knowledge about its environment becomes available, and uses these skills in an active learning setting to speed the discovery of unknown structure, in turn allowing for the construction of new skills, and so on. This active learning is realized through the use of an intrinsic reward function that guides the agent to areas of the state space for which its current knowledge about the dynamics of its environment is lacking, but that it can reliably reach. The bootstrapped nature of this approach to learning leads to acquisition of complex behaviors not achievable by simpler methods.

1.2.2 Intrinsically Motivated Skill Learning in Factored MDPs

We extend existing work on hierarchical decomposition of finite factored MDPs (FMDPs) and active learning of their dynamical structure to apply the framework described in the previous contribution to FMDPs, which brings to bear the benefits of leveraging environmental structure on skill learning in more complex environments. These methods and their advantages in learning solutions to ensembles of related tasks are demonstrated in a large, factored domain called the "light box".

1.2.3 Incremental Structure Learning in Continuous Factored MDPs

While the previous contribution makes use of existing work on structure learning in finite FMDPs, these techniques do not readily apply to continuous FMDPs. We present a novel algorithm for online, incremental learning of transition models for factored MDPs that have continuous, multi-dimensional state and action spaces. We use incremental density estimation techniques and information-theoretic principles to learn a factored model of the transition dynamics of a continuous FMDP online from a single, continuing trajectory of experience.

1.2.4 Temporal Difference Networks for Continuous Dynamical Systems

Temporal-difference (TD) networks are a class of predictive state representations that use well-established TD methods to learn models of partially observable dynamical systems. Previous research with TD networks has dealt only with dynamical systems with finite sets of observations and actions. We present an algorithm for learning TD network representations of dynamical systems with continuous observations and actions. We show that the algorithm is capable of learning accurate and robust models of several noisy continuous dynamical systems.

1.3 Organization of Thesis

In the following chapter, we outline the relevant background material and previous work in machine learning related to learning hierarchical solutions to sequential decision problems in MDPs and structured environments like FMDPs. In Chapter 3, we present a framework for long-term, incremental, intrinsically motivated learning of skill hierarchies in MDPs and show its benefits to learning solutions to ensembles of tasks in complex environments. Chapter 4 contains an extension of this framework to structured environments and presents a novel solution to intrinsically motivated skill learning in FMDPs. Chapters 5 and 6 address methods for model learning in environments that don't satisfy the same properties as MDPs and finite FMDPs. In particular, Chapter 5 describes an algorithm for incremental structure learning of transition models in continuous FMDPs, while Chapter 6 describes an algorithm for online learning of TD networks as models of continuous dynamical systems. Finally, in Chapter 7, we summarize the contributions of this work, outline their limitations, and discuss potential future work.

CHAPTER 2
BACKGROUND AND RELATED WORK

This chapter contains a detailed outline of the relevant background material and related work in machine learning pursuant to the contributions we present in later chapters. We begin by giving a brief overview of Markov Decision Processes (MDPs) and the computational reinforcement learning framework. We then discuss the options framework, a formalism for hierarchical skill learning in reinforcement learning agents, and the advantages options provide when learning and planning in MDPs. We also outline recents developments in the options framework, referred to as universal option models, which allow for models of skills to be factored into two components: one that predicts a skill's dynamical effects on its environment, and another that predicts the rewards expected during execution of the skill. As we show in Chapter 3, this factorization allows for more efficient planning by an agent maximizing intrinsic rewards in the framework presented therein.

Subsequent sections deal with a variation of MDPs called factored MDPs (FMDPs), which allow for exploitation of environmental structure not possible with traditional MDP representations. We review recent work in information-theoretic structure learning of FMDP models, a technique used in the framework presented in Chapter 4. We continue with a discussion of applications of hierarchical reinforcement learning to solving FMDPs more efficiently. Finally, we conclude this chapter with an overview of the intrinsically motivated reinforcement learning literature and some open questions which we address in later chapters.

2.1 Reinforcement Learning and Markov Decision Processes

Reinforcement learning (RL) methods are a class of optimization techniques that search for optimal solutions to sequential decision problems (Sutton & Barto, 1998). An RL problem is defined with respect to an agent which interacts with an environment by taking actions and receiving observations reflecting the effects of its actions on the environment. The agent also receives a scalar-valued reward signal, some function of which it attempts to maximize over time by selecting appropriate actions. An RL agent's environment is often formalized as a Markov Decision Process (MDP).

An MDP is a tuple $\langle \mathcal{S}, \mathcal{A}, p, r, \gamma \rangle$, where $\mathcal{S}$ is a set of states, $\mathcal{A}$ is a set of actions, $P : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow [0, 1]$ is a one-step transition function which specifies a probability distribution over successor states given a current state and action, $R : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow \Re$ is an immediate reward function which determines the real-valued reward an agent receives for taking a given action in a given state and transitioning to a given successor state, and γ is a discount factor explained below. An MDP is assumed to satisfy the Markov property, which guarantees that the one-step models R and P are sufficient for predicting the distribution of rewards and successor states any number of time steps in the future given a current state and sequence of actions.

When the task of an RL agent is formulated as an MDP, the objective of the agent is usually defined to be the maximization of its expected discounted sum of future rewards, or *expected discounted return* . The discounted return at time t is defined as

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}, \tag{2.1}$$

where $\gamma \in [0, 1]$ is a discount factor that determines the degree to which more immediate rewards are preferred over more distant ones. If $\gamma = 0$, only immediate rewards are taken into account when making decisions. As γ approaches 1, the agent considers rewards arbitrarily far into the future when choosing actions.

A solution to an MDP is a function $\pi : \mathcal{S} \times \mathcal{A} \rightarrow [0, 1]$, called the agent's *policy*, where $\pi(s, a)$ gives the probability of selecting action $a \in \mathcal{A}$ in state $s \in \mathcal{S}$. The goal of an agent is to maximize (2.1) by finding an optimal policy π^*, which (probabilistically) takes actions that maximize (2.1) in every state. Many RL algorithms maintain an estimate of (2.1) in the form of a *value function* and use this estimate to guide the search for an optimal policy.

A state-value function $V^\pi : \mathcal{S} \rightarrow \Re$ maps states to real numbers representing the expected discounted return for starting in a given state and following policy π. An action-value function $Q^\pi : \mathcal{S} \times \mathcal{A} \rightarrow \Re$ maps state-action pairs to real numbers representing the expected discounted return for starting in a given state, taking a given action, and from then on following policy π. Thus, for a given $s \in \mathcal{S}$ and $a \in \mathcal{A}$, $V^\pi(s) = E_\pi[R_t | s_t = s]$ and $Q^\pi(s, a) = E_\pi[R_t | s_t = s, a_t = a]$. These functions represent how good being in a given state or taking a given action in a given state is under policy π, where good is defined in terms of expected return.

Value functions can be expressed recursively in the form of *Bellman equations*, which relate the value of a state to the values of its successor states. The value function under a given policy π can be written as the Bellman equation

$$V^\pi(s) = \sum_a \pi(s, a) \sum_{s'} P(s, a, s') \Big[R(s, a, s') + \gamma V^\pi(s') \Big]. \tag{2.2}$$

Optimal value functions, those defined with respect to an optimal policy π^*, can also be expressed recursively using the Bellman optimality equation

$$V^*(s) = \max_a \sum_{s'} P(s, a, s') \Big[R(s, a, s') + \gamma V^*(s') \Big]. \tag{2.3}$$

Simliar Bellman equations exist for action value functions Q^π and Q^*.

Value functions can aid in the learning of optimal policies through *policy iteration*, an iterative technique that alternates between evaluating the agent's current policy (estimating its value function) and improving the policy based on the evaluation.

The evaluation of π is performed by estimating V^π from experience executing π. The policy is then improved in some way according to the most recent evaluation. Often the improvement is done by altering the policy so as to greedily select actions that maximize V^π in each state. Improvement produces a new policy π' and corresponding $V^{\pi'}$, and the process repeats until $\pi = \pi'$.

When the transition function P is known or estimated from experience, *model-based* RL can be employed to expedite value function learning in the sense of requiring less experience for V^π to converge to the optimal value function V^* (Sutton, 1991). If the reward function is also known or estimated, *value iteration* can be used to compute an optimal value function and corresponding policy directly by turning the Bellman optimality equation into an update rule for a sequence $\{V_k\}$ of successive approximations:

$$V_{k+1}(s) = \max_a \sum_{s'} p(s,a,s') \Big[r(s,a,s') + \gamma V_k(s') \Big], \tag{2.4}$$

which can be shown to converge to V^* under certain conditions.

Even when model-based methods are used in this way to improve data efficiency, tabular representations of value functions and policies (i.e., those with one entry per state or state-action pair) become infeasible to learn or compute efficiently for large MDPs. For this reason, much work has focused on approximation techniques that allow for both generalization of value between similar states, and compact representations of value functions.

One class of these approximation methods is appropriate when the transition and reward functions of the MDP can be represented in factored form, affording the potential for certain dimensions of the MDP to be irrelevant when predicting the effects of actions on other dimensions. In these cases, this structure can be exploited to learn or compute compact representations of value functions and policies more efficiently (Boutilier et al., 2000). This leads to the factored MDP formalism, which we discuss in detail in Section 2.4.

Another class of approximation techniques, known as *linear function approximation*, defines an estimate $\hat{V}^\pi$ of value function V^π as a weighted sum of basis functions (sometimes called features) $\phi_i : \mathcal{S} \rightarrow \Re$:

$$\hat{V}^\pi(s) = \vec{\theta}^{\,T}\vec{\phi}(s) = \sum_i \theta_i \phi_i(s), \tag{2.5}$$

where $\vec{\theta}$ is a weight vector with size equal to the number of basis functions $|\vec{\phi}(s)|$. Learning an approximation thus entails finding a $\vec{\theta}$ that minimizes the error between $\hat{V^\pi}$ and V^π.

When value functions are represented in this way, they are said to be linear in the parameters $\vec{\theta}$, which allows for some mathematical conveniences. In particular, standard gradient-descent methods can be used straightforwardly to learn value functions of this sort, since the gradient of the approximate value function with respect to $\vec{\theta}$ is simply

$$\nabla_{\vec{\theta}} \hat{V}^\pi(s) = \vec{\phi}(s). \tag{2.6}$$

Additionally, any gradient descent method guaranteed to converge to or near a local optimum in this linear case is automatically guaranteed to converge to or near a global optimum.

It is worth noting that this representation for linear function approximation can be defined so as to include tabular representations as well. In this case, $\phi(s)$ is simply a vector of size $|\mathcal{S}|$ that contains a 1 for the element corresponding to s and zeroes for all other entries. When ϕ is represented this way, the update rules for learning value functions become equivalent to the corresponding tabular update rules, since the gradient of the value function in this case become a selector for the current state.

When ϕ is not defined to provide a one-to-one mapping from states to features in this way, however, linear function approximation affords the application of reinforcement learning techniques to larger MDPs than would be feasible with tabular representations partly because it allows for generalization of value between similar states, eliminating the need to visit a certain state to improve an estimate of its value. Although this is an important part of applying RL to large problems, it doesn't address the difficulty of learning policies when long, specific sequences of actions are needed to reach certain areas of the state space. Additionally, when planning in MDPs using methods like value iteration, obtaining the value of long sequences of actions becomes computationally prohibitive quickly. One way to mitigate this is to reason about the effects of action sequences at different time scales using temporally extended actions. The field of hierarchical reinforcement learning addresses formalisms for this type of temporal abstraction.

2.2 Hierarchical Reinforcement Learning

The *options* framework is a formalism for temporal abstraction in RL that details how to learn and use temporally extended actions in MDPs (Sutton et al., 1999). An option is a closed-loop controller defined as a tuple $\langle I, \pi, \beta \rangle$, where $I \subseteq S$ is a set of states over which the option is defined (the initiation set), π is the policy of the option, defined over I, and $\beta : S \rightarrow [0, 1]$ is a termination condition function that gives the probability of the option terminating in a given state. Options can also be understood as sub-MDPs embedded within another (possibly larger) MDP. As such, all of the machinery associated with learning solutions to MDPs also applies to learning options, with some subtle differences.

To be useful in planning, models of the long-term effects of options must be learned as well. An *option model* $\langle R^o, P^o \rangle$ is comprised of two parts: a reward model and a transition model, which are analogues of the reward and transition functions of an

MDP. The reward model of an option o gives the expected discounted reward received after executing o in state s at time t and following o's policy until termination:

$$R^o(s) = \mathbb{E}_{s,o}[r_t + \gamma r_{t+1} + \cdots + \gamma^{T-1} r_T], \tag{2.7}$$

where T is the random termination time of o. The transition model of an option o gives the discounted probability of terminating at a state s' given that o is executed from state s and its policy followed until termination:

$$P^o(s, s') = \sum_{k=1}^{\infty} \gamma^k p(s', k), \tag{2.8}$$

where $p(s', k)$ is the probability that o terminates at s' after k steps. Algorithms for learning the policy, reward model, and transition model of an option from experience with an MDP are given in Sutton, Precup, and Singh (1999).

The advantage of estimating the transition and reward models of an option is that the option can be treated as a primitive action in model-based RL methods. This means that algorithms like value iteration can propagate the value of executing long sequences of primitive actions in just one iteration, which allows an agent to plan far into the future with limited computational resources. Additionally, since options can call other options in their policies, agents can construct deeply-nested policies with multiple levels of behavioral abstraction, leading to increased efficiency in both learning and planning as the hierarchy deepens. We make use of these key properties of options in the frameworks presented in Chapters 3 and 4.

Although much attention has been devoted to learning options in MDPs, most of these approaches use the same state representation for every option, leading to temporal abstraction but not state abstraction. Less research has focused on learning options in FMDPs, where it is possible for different options to have different representations. Section 2.6 discusses the relevant work involved in constructing options in finite FMDPs, each with its own state abstraction, a technique we make use of in Chapter 4. The following section, however, discusses a recent advancement in the options framework that allows for factorization of option models into independent transition and reward components. This result has great utility in the framework presented in Chapter 3.

2.3 Universal Options

Learning the reward model of an option from experience results in a model that is specific to the particular extrinsic reward function of the MDP in which the option is embedded. While this may be useful for single tasks, it is less useful in the scenario of interest in this thesis, wherein we would like an agent to reuse an option as part of solutions to many tasks, each of which may have its own distinct reward function. The limited utility of a traditional option model stems from the fact that it can only be used to reason about the effects of executing that option in the context of the reward function under which it was learned. Far better is an option model that can be used to reason about the option's effects independent of a specific reward function. Representations of such reward-independent options models, and techniques for learning them, were proposed recently by Yao, et al. (2014).

Universal option models (UOMs) represent the reward model of an option as the composition of a reward-independent function and a reward function. The reward-independent function need only be learned once, and can subsequently be composed with any valid reward function to produce a reward model that can be used in planning techniques such as value iteration. Yao, et al. (2014) show that the resultant reward model behaves exactly like a traditional option reward model as if it had been learned under the composed reward function.

Formally, the reward model, R^o, of a UOM for option $o = \langle I, \pi, \beta \rangle$ is defined as the composition of a *discounted state occupancy function,* u^o, and a reward vector, r^π, under the option's policy:

$$R^o(s) = \sum_{s' \in \mathcal{S}} r^\pi(s') u^o(s, s'). \tag{2.9}$$

The discounted state occupancy function is defined as

$$u^o(s, s') = \mathbb{E}_{s,o}\Big[\sum_{k=0}^{T-1} \gamma^k \mathbb{I}_{\{s_k = s'\}}\Big], \tag{2.10}$$

where $\mathbb{I}_{\{\cdot\}}$ is the indicator function, which equals 1 when its condition is satisfied and 0 otherwise. The transition model, P^o, of a UOM is identical to the transition model of a traditional option, and thus still defined as (2.8).

Yao, et al. (2014) also develop a formalism for linear UOMs, which represent UOMs using linear function approximation, and thus allow their application to continuous domains. Given a feature representation like that described in Section 2.1, which maps a state $s \in \mathcal{S}$ to an n-dimensional feature vector $\phi(s)$, a linear UOM for an option o is defined as a pair of $n \times n$ matrices (U^o, M^o) which generalize, and are backwards compatible with, the UOM functions (u^o, P^o) in the tabular representation discussed above.

Given a reward function $\mathcal{R}$, one can learn a least-squares approximation $f^{(LS,\mathcal{R})}$ of the reward function in terms of the feature representation. With this approximation, one can show that $f^{(LS,\mathcal{R})^\top}(U^o\phi)$ is a valid approximation of the expected return R^o of option o. Similarly, given a feature vector ϕ, $M^o\phi$ predicts the discounted expected feature vector in which the option terminates. Thus, in the linear function approximation case, the U^o matrix provides the reward-independent component of the option's reward model and the M^o matrix the transition model. Incremental methods for learning these matrices from experience executing an option are given in Yao, et al. (2014).

The UOM mechanism for reasoning about the effects of options without needing to learn a separate option model for each unique reward function provides a critical advantage to an agent in our framework, whose intrinsic reward function will change frequently. We discuss the details of our framework and where this advantage comes from in Chapter 3. In the following sections, however, we shift gears to discuss the necessary background material for the approach presented in Chapter 4, which presents an extension of our framework to environments where statistical structure can be leveraged to learn and plan with more compact representations of options.

2.4 Factored MDPs

A factored MDP (FMDP) is an MDP in which the state space $\mathcal{S}$ is defined as the Cartesian product of the domains of a finite set of random variables $\{S_1, \dots, S_n\} = \mathbf{S}$. States in FMDPs are thus represented as vectors—assignments of specific values to the variables in $\mathbf{S}$. For ease of exposition, also assume for now that each variable $S_i \in \mathbf{S}$ is a binary random variable so that the domain $Dom(S_i)$ of S_i is $\{0, 1\}$, though all of the methods discussed here hold for the multinomial case as well. The FMDP formalism can also be used to represent environments with continuous state and action variables, though the work outlined in this chapter and the framework presented in Chapter 4 only cover the case in which the action set is finite and the state variables take on discrete values. Chapter 5 presents the details of the formalism in the continuous case.

The transition function P of a finite FMDP is often represented as a set of dynamic Bayesian networks (DBN), one for each action (Dean & Kanazawa, 1989). In this case, a DBN is a two-layer directed acyclic graph with nodes in layers one and two representing the variables of the FMDP at times t and $t+1$ (see Figure 2.1). Edges in a DBN represent dependencies between variables. We make the common assumption that there are no synchronic arcs in each DBN, meaning that variables within the same layer do not influence each other.

To simplify notation, let S_i and S_i' represent the random variable $S_i \in \mathbf{S}$ at times t and $t+1$ respectively, and let $s_i, s_i' \in Dom(S_i)$ denote specific instantiations of those

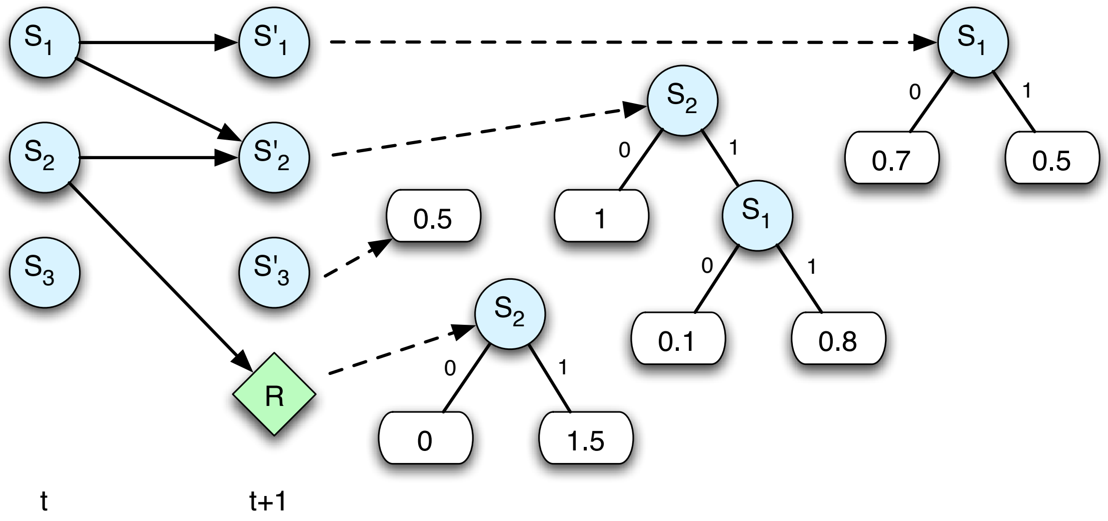

Figure 2.1. An example DBN for a given action a. Each decision tree represents a conditional probability distribution for its associated random variable (indicated by dashed arrows). Leaves show the probability that the tree's associated variable is 1 at time $t+1$ given the values of its parents at time t (shown as labels along the leaf's branch).

random variables. Furthermore, let $f_{\mathbf{X}}(\mathbf{s})$ denote the projection of a state vector $\mathbf{s} \in \mathcal{S}$ onto the set of variables $\mathbf{X} \subseteq \mathbf{S}$; that is, the values of the variables in $\mathbf{s}$ corresponding to the variables in $\mathbf{X}$. Finally, let $Par(S'_i, a)$ denote the set of parents of the random variable S'_i given by the sources of the incoming links to variable S'_i in the DBN for action a. When an FMDP is represented as a DBN in this manner, the transition function P can be expressed in factored form as:

$$P(\mathbf{s}'|\mathbf{s}, a) = \prod_i^n P(S'_i | f_{Par(S'_i,a)}(\mathbf{s})). \tag{2.11}$$

Intuitively, this says that the probability of transitioning to state $\mathbf{s}'$ from state $\mathbf{s}$ when taking action a at time t can be represented as the product of the conditional probabilities of each state variable at time $t+1$ given the projection of state $\mathbf{s}$ onto that variable's parents. The individual conditional probability distributions in the product of (5.1) can be compactly stored as conditional probability trees (CPT), each of which contains internal nodes corresponding to the parents of S'_i (under action a) and leaves containing a probability distribution over $Dom(S_i)$, as shown in Figure 2.1. The labels along the branch of each leaf provide the appropriate conditioning values. The expected reward can be modeled in a similar way, and is represented by the diamond-shaped node in Figure 2.1. The conditional expectation of immediate reward can also be represented by a tree with leaves containing expected reward given the conditioning values along the leaf's branch.

When the transition dynamics of an FMDP contains relatively sparse inter-variable dependencies, this factorization can have a dramatic effect on the computation of value functions and optimal policies by reducing the effect of the curse of dimensionality (Bellman, 1957), although this is not guaranteed. There has been a considerable amount of work on reinforcement learning and planning algorithms that exploit this structure when the transition and reward models are given (Boutilier et al., 2000). One of these methods, structured value iteration (SVI), is a version of value iteration that exploits this structure to reduce the computation involved in calculating an optimal value function (and corresponding policy). We make use of SVI in the framework presented in Chapter 4. Other work has focused on efficient online learning of the transition model so as to make these algorithms appropriate when the model is not known in advance. We describe these approaches in the following section.

2.5 Incremental, Active DBN Structure Learning

When the transition model of an FMDP is represented as a set of DBNs, learning the model from experience amounts to learning the structure and parameters of a Bayesian network. The problem of Bayesian network structure learning is to find the network $B = \langle G, \theta \rangle$ that best fits a data set $\mathbf{D}$, where G in our case represents the graphical structure of a DBN, θ represents the corresponding CPTs, and data points are in the form of state-successor pairs $\langle \mathbf{s}, \mathbf{s}' \rangle$. Although the literature on Bayesian network structure learning is substantial, many of these methods are not incremental and generally require that the data are drawn in an i.i.d. fashion (Abbeel et al., 2006). When attempting to learn the structure of a DBN online from experience with an FMDP, the scenario of interest in this thesis, these methods are thus not applicable. There are, however, a few incremental methods developed recently which search for DBN structures that fit an agent's experience with an FMDP well, where the data are not drawn i.i.d. because of the temporal dependencies involved. We make use of an approach given in Jonsson and Barto (2007) in our framework, which we outline below. Alternative approaches, along with their advantages and disadvantages, are discussed in Section 4.6.

Jonsson and Barto (2007) present a greedy, incremental structure-learning method that uses the Bayesian Information Criterion (BIC) to evaluate potential splits at the leaves of the CPTs which are learned. Their focus is primarily on structure learning, and so they do not embed their approach in a reinforcement learning scheme as in Degris et al. (2006), though such an embedding is a component of the contributions presented in Chapter 4. Rather, their contribution is an *active* learning mechanism by which structure can be learned more quickly than if random policies or exploitation-biased policies are used to collect data. This concept of active structure learning is a key component of the contributions of this thesis.

When a learning agent has some degree of control over the data samples it receives during training, the agent may engage in what is called *active learning*, in which it requests training examples from a teacher or its environment that result in faster learning when compared with training data selected at random or by the teacher (Cohn et al., 1994). Although an arbitrary policy (e.g., random) could be used to collect the data necessary to learn DBN structure in FMDPs, it is interesting to consider exploration policies that attempt to maximize the rate at which this structure is learned, thus allowing an agent to engage in active learning of that structure.

Recall that the goal of structure learning in this context is to find a best-fit Bayesian network $B = \langle G, \theta \rangle$ given a data set $\mathbf{D}$. One way to find such a network is to compute the posterior probability distribution $P(G|\mathbf{D})$ over a set of networks and choose the one that maximizes this distribution. While is not feasible to compute this distribution directly, there are approximation techniques that have been shown to perform well. It follows from Bayes theorem that $P(G|\mathbf{D}) \propto P(\mathbf{D}|G)P(G)$. One approximation technique, known as the Bayesian Information Criterion (BIC), makes the approximation

$$\log[P(\mathbf{D}|G)P(G)] \approx L(\mathbf{D}|G) - \frac{|\theta|}{2} \log |\mathbf{D}|, \tag{2.12}$$

where $L(\mathbf{D}|G)$ is the log-likelihood of the data given the network (Schwarz, 1978). When all data values are observable, this likelihood can be decomposed as

$$L(\mathbf{D}|G) = \sum_i \sum_j \sum_k N_{ijk} \log \theta_{ijk}, \tag{2.13}$$

where N_{ijk} is the number of data points $x \in \mathbf{D}$ such that $f_{\mathbf{Pa}(S_i^{t+1})}(\mathbf{s}^t) = j$ and $f_{\{S_i^{t+1}\}}(\mathbf{s}^{t+1}) = k$, and $\theta_{ijk} = P(S_i^{t+1} = k|\mathbf{Pa}(S_i^{t+1}) = j)$. This quantity is maximized for $\theta_{ijk} = N_{ijk}/\sum_k N_{ijk}$. Although finding the network with the best BIC score is known to be NP-complete (Chickering et al., 1995), the score decomposes into a sum of terms for each variable S_i and each value of j and k that only changes locally when edges between variables are added or deleted. One can thus incrementally add or delete edges greedily to find high-scoring (though possibly sub-optimal) networks.

Before explaining the intuition behind the approach taken by Jonsson and Barto (2007), who use the BIC metric as a means for evaluating network structure, we elaborate on their method for learning the CPTs that define an FMDP's transition function. For each variable-action pair in an FMDP, a CPT is maintained (initially consisting of a single leaf) which stores data points of the form $\langle \mathbf{s}, \mathbf{s}' \rangle$, each assigned to the appropriate leaf based on the variable assignments given by $\mathbf{s}$. Each time a new data point is added to a leaf, the BIC score of the data at the leaf and the scores associated with each possible refinement of that leaf are computed. A *refinement* of a leaf l is a split of l on some variable S_j, resulting in a new child leaf for each value in $Dom(S_j)$, to which the data instances of l are distributed accordingly. If the sum of the BIC scores associated with any refinement of a leaf is greater than the current BIC score of that leaf, then the refinement is kept. Refinements of a leaf l on a variable S_j are not considered if S_j is already on the path from the root of the tree to l. Only refinements at non-empty leaves that have collected at least k samples for each possible split variable are considered, where $k \in \mathbb{Z}^+$ is a parameter that informally determines the level of confidence in the accuracy of the network's refinements.

With this framework for structure learning established, Jonsson and Barto (2007) incorporate active learning by having an agent choose a primitive action at every step in order to maximize the sum of the entropies of the distribution vectors that are analyzed when considering refinements at leaves of the CPTs. By doing this, each leaf of each CPT collects samples in a more uniform fashion over its potential split variables than happens via random action selection, resulting in quicker refinement evaluations and consequently faster structure learning. The algorithm looks at the current state of the environment and, for each action, determines to which leaf the resulting transition sample would map for each CPT associated with that action. The associated change in entropy of each distribution vector at each of those leaves is calculated for each action and the action with the largest total change in entropy is selected with probability $1 - \epsilon$, where $0 < \epsilon < 1$ is a random exploration parameter. Otherwise, a random action is selected.

Although this approach does produce faster learning in some domains, in more complex domains the approach still fails to discover a significant portion of the environmental structure (Jonsson & Barto, 2007). This is because the algorithm is myopic, only considering the effects of primitive actions at each step, and thus easily becomes stuck in areas of the state space that are difficult to get out of without explicit planning. This limitation can be remedied by introducing temporally abstract actions into an agent's action set, which allow for longer-term planning to reach configurations of domain variables that will yield more relevant information about the environment. Chapter 4 presents a novel approach to this extension, but we first outline the relevant work in temporal abstraction and hierarchy in RL, and their application to planning and learning in FMDPs.

2.6 Hierarchical Decomposition of FMDPs

Jonsson and Barto (2006) present a framework for option discovery and learning in finite FMDPs. The *variable influence structure analysis* (VISA) algorithm discovers options by analyzing the causal graph of a domain, which is constructed from the dependencies exhibited in the DBNs that define the FMDP. There is an edge from S_i to S_j in the causal graph if there exists an edge from S_i^t to S_j^{t+1} in the DBN model for any action. The algorithm identifies in the causal graph context-action pairs, called *exits*, that cause one or more variables to change value when the given action is executed in the corresponding context (a set of variable-value pairs). By searching through the conditional probability distributions that define the DBNs, exit options are then constructed to reliably reach this context from any state and execute the appropriate action. The agent's overall task is then decomposed into sub-tasks solved by these options. VISA takes advantage of environmental structure to learn compact policies for options by ignoring irrelevant variables. Each option's value function and policy are defined only over variables relevant to achieving the goal of the option, resulting in (often significant) option-specific state abstraction.

Another feature of the framework is a method for computing compact option models from a given DBN model. The models are compact in that they take the same form as the models of primitive actions (DBNs) and represent with decision trees the probability distributions over the variables of the FMDP expected once the option finishes executing from a given state. Having option models in this form allows for their use in planning as atomic actions. This also means that one can use SVI to compute new option policies in terms of existing options very efficiently. This will form the basis of the hierarchical active learning scheme we present in Chapter 4.

2.7 Intrinsically Motivated Reinforcement Learning

Intrinsically motivated behavior has been described as behavior that is rewarding for its own sake, rather than because it fosters progress in solving a specific problem (Barto et al., 2004). The psychology literature has shown that humans and other mammals, especially young ones, often engage in intrinsically motivated behavior when they are not preoccupied with survival or reproductive goals (Harlow et al., 1950). Although intrinsically motivated behavior may not be immediately motivated by survival, engaging in it seems to confer significant survival advantages over organisms that do not. This may be because such behavior promotes the acquisition of increasingly complex abstract skills, which can be readily applied to novel problems later in life. An organism without such a set of skills will be ill-equipped to handle these challenges as compared to an organism with a rich library of reusable behaviors.

The incentive for modeling intrinsic motivation in artificial RL agents is to produce agents that are motivated to learn complex hierarchies of skills applicable to the solution of a broad range of problems in a given environment. Such agents would be equipped to solve many instances of related problems, not just one. This direction of research is considerably different from traditional RL approaches, which are generally concerned with efficient learning of solutions to individual sequential decision problems. One consequence of this focus is the need for an extensive period of exploration in which an agent can acquire and perfect a rich set of skills. During this period the agent may be solely motivated by acquisition of these skills, potentially unaware of the types of problems it may be called upon to solve at a later time.

Early work on intrinsic motivation focused exclusively on efficient learning of world models in sequential decision problems, and were not specifically concerned with skill learning. These approaches provided intrinsic reward to agents proportional to errors in the predictions of their world model, leading the agent to areas of the environment which are unpredictable, thereby focusing learning on those areas so as to reduce that unpredictability (Schmidhuber, 1991). In stochastic environments, however, this causes the agent to become "obsessed" with inherently unpredictable regions, since they provide high reward indefinitely. Thus, methods that reward agents for *progress* in improving model quality were proposed, causing agents to become "bored" with such inherently unpredictable areas (as well as predictable ones), since they afford no learning progress (Kaplan & Oudeyer, 2004; Schmidhuber, 2005). These methods, however, we also largely focused on model learning, and not skill acquisition.

Barto et al. (2004) were the first to suggest intrinsic motivation as a method for driving the accumulation of hierarchical skill sets, and proposed an intrinsic reward mechanism that encouraged agents to build skills which reliably cause certain (pre-specified) salient events to occur. Simsek and Barto (2006) generalized this somewhat and presented an algorithm that rewards the agent for improvements in the value function of a given task or option, which they show can speed up learning of that value function by focusing exploration on areas where learning will have the most influence. None of these approaches to intrinsically motivated skill learning, however, employ model-based RL or use the agent's existing skill set to bootstrap learning of new skills, which we argue is an essential aspect of developmental learning.

The framework presented in Chapter 3 combines the techniques discussed above and employs them in a truly developmental curriculum to achieve this type of lifelong learning in agents whose environments can be represented as MDPs. In Chapter 4, we present a second framework that extends this approach to the case of structured environments using the techniques for structure learning and skill learning in FMDPs mentioned above. This framework takes advantage of the state abstractions afforded by factored representations to achieve increased efficiency of skill learning in such environments.

CHAPTER 3
INTRINSICALLY MOTIVATED SKILL LEARNING

In this chapter, we present a novel algorithm for intrinsically motivated, developmental learning of skill hierarchies in RL agents whose environments can be represented formally as MDPs. The approach draws on several of the formalisms and techniques discussed in the previous chapter and integrates them into a cohesive algorithm for motivating agents to incrementally improve both their understanding of how their actions affect their environment, and their ability to manipulate that environment. Recall that the primary objective of this approach is to produce agents that are motivated to learn a hierarchical collection of abstract skills that may be used independently and in combination with each other as solutions to common subproblems encountered in novel tasks later in life. This process of skill learning must be incremental and bootstrap the agent's skill acquisition with its current skill set and models of environmental dynamics, but not require an explicit curriculum from an external teacher.

In the following sections, we discuss some prerequisites before describing the full algorithm. These include how skills are represented and created, how internal knowledge state is leveraged to generate intrinsic rewards, and the types of domains in which the framework will confer significant advantages over non-developmental approaches to autonomous skill learning. We then present the details of the algorithm for both finite and continuous-state MDPs, and show its performance on environments that illustrate its strengths. Finally, we discuss related work, the limitations of this approach, and directions for future work.

3.1 Skill Representation

Agents employing our algorithm must have a way to learn, and plan with, temporally abstract actions, or skills. We adopt the options formalism, described in Section 2.2, as our representation of skills. Recall than an option, $\langle I, \pi, \beta \rangle$, consists of an initiation set $I \subseteq S$ in which the option can be initiated, a policy π defined over the initiation set, and a termination function $\beta : S \rightarrow [0, 1]$ that determines the probability of an option terminating in a given state. Also recall that models of options, $\langle R^o, P^o \rangle$, consist of a reward model, $R^o(s)$, which gives the expected discounted return of executing an option in a given state, and an option transition model, $P^o(s, s')$, which gives the expected discounted probability distribution over successor states after executing an option in a given state. These models can be learned from experience and used in algorithms like value iteration to perform planning in MDPs.

Of course the use of options requires that their data structures be in place in order to learn about their effects. This means that an agent must have some policy for creating new options. The problem of appropriate and efficient option discovery is an open problem in hierarchical RL. There have been several approaches proposed (Simsek, 2008; Hart et al., 2008; Konidaris, 2011), each based on different performance metrics or analyses of environmental structure. Although this is an interesting and critical problem, we do not focus on its solution in this work. Rather, we employ a heuristic solution to option creation for each domain in which we evaluate our algorithm, leaving the option discovery problem to future work. We note, however, that any method for option discovery can be used in conjunction with our algorithm as long as it can identify subgoals incrementally; i.e., without knowledge of domain dynamics prior to the agent experiencing them. Additionally, although options are expressive enough to represent continuing skills which do not necessarily have goal states (e.g., walking or balancing), we restrict our focus in this work to subgoal options—options whose optimal policy is to reach a given state or set of states and then terminate.

We use the formalism discussed in Section 2.3 on UOMs to represent option models in our framework. Specifically, we use linear UOMs for both finite and continuous MDPs, since the formalism for linear UOMs supports both equivalently. Recall that a linear UOM consists of two $n \times n$ matrices, U^o and M^0, where n is the number of states of the MDP in the tabular (finite) case, and the dimensionality of the feature representation, ϕ, in the continuous case. The U^o matrix is a reward-independent operator that can be composed with an immediate reward vector $\mathbf{r}$ of dimension n to produce an expected return model for the option; i.e., $R^o(s) = (\mathbf{r}^\top)U^o\phi(s)$. The M^o matrix is a transition operator that gives the expected discounted successor state (feature vector) when executing the option from a given state, given by $M^o\phi(s)$. Incremental methods for learning the matrices U^o and M^o from experience are given in Yao, et al. (2014).

With this formalism as our choice of skill representation, we now have a mechanism by which, given immediate rewards $\mathcal{R}^a$ and their corresponding feature vector representation $\mathcal{R}^a_\phi$, we can compute value functions, and consequently policies, using standard dynamic programming techniques like value iteration that incorporate the long-term effects of options. Moreover, we can change the reward function as often as needed without the need for new learning, since the option models are reward independent. As we discuss in the following section, we will use this fact to generate intrinsic reward functions on-the-fly as the agent explores and updates the accuracy of its environmental model.

3.2 Intrinsic Reward Functions

There are many possibilities for intrinsic reward functions that encourage the acquisition of skill hierarchies in RL agents. In general, since these functions are *intrinsic* reward functions, it makes sense that they should incorporate some aspect of an agent's internal state, which could incorporate factors such as innate preferences, novelty, habituation, predictive confidence, and so on. The intrinsic rewards we employ in this work are a function of an agent's confidence in its model of the effects of its actions on the environment. This choice is motivated in part by previous work by other researchers investigating computational models of intrinsic motivation, as discussed in Section 2.7, and in part by the techniques and formalisms we have outlined thus far, which provide mechanisms for turning models of abstract skills into plans and policies for tasks that make use of those skills.

In order to generate intrinsic rewards as a function of an agent's confidence about the accuracy of its model of the environment, there must be some formal representation of this confidence. Since the models of skills our algorithm uses are learned directly from experience, and the goal of an agent in our approach is to learn skills to manipulate its environment as efficiently as possible using these learned models, we choose a formalism that defines confidence as a function of the number of samples an agent has observed for taking an action a in a given state s. Intrinsic reward is then defined as the inverse of this confidence, making state-action pairs that have been visited infrequently more rewarding than those that have been visited often.

More formally, an agent's immediate intrinsic reward function under action a, $r_I^a : \mathcal{S} \rightarrow \Re$, is defined as a linear function of the agent's feature set ϕ:

$$r_I^a(s) = \vec{\phi}(s)^T \vec{\rho^a}, = \sum_i \phi_i(s)\rho_i^a. \tag{3.1}$$

where ρ^a represents a vector of weights on the features of ϕ, so that $|\vec{\rho}| = |\phi|$. These weights are adjusted accordingly to reflect model confidence as an agent performs their corresponding actions in various states. Specifically, when an agent executes action a_t in state s_t at time step t, the weights ρ_t^a are updated according to the update equation

$$\rho_{t+1}^{a_t} \leftarrow \rho_t^{a_t} - \zeta \phi_i(\mathbf{s}_t)^T \rho_t^{a_t}, \tag{3.2}$$

where $\zeta < 1$ is a step-size parameter.

The weights are initialized to $\vec{1}$ (a vector of all ones) for each action, making all state-action pairs initially maximally rewarding. This update rule thus has the effect of decreasing $r_I^a(s)$ exponentially towards zero as the agent continues to execute a in s. In continuous environments, when ϕ is chosen in a way that allows for generalization (e.g., a set of radial basis functions), this change in intrinsic reward will generalize to similar states as well.

The goal of this choice of intrinsic reward function is to motivate an agent to spend its time collecting data in areas of the state space about which it currently has low confidence in the accuracy of its environmental model. As more data is collected, and its model confidence increases, the intrinsic rewards for those state-actions pairs will decay exponentially, becoming less interesting to the agent, and encourage it to explore areas of the state space where its model confidence is lower. While this initialization scheme may seem to encourage the agent to focus exploration on states that it does not know how to reach, we will see in the following section, which provides the full specification of our algorithm, that our method for computing the agent's exploration policy will reduce this effect and keep the agent exploring areas right at the fringe of its expertise.

3.3 Algorithm

Algorithm 1 gives pseudocode for the main loop of our algorithm for intrinsically motivated learning of skill hierarchies in MDPs. We outline the steps of the algorithm here to provide intuition for its expected behavior. We present the algorithm for the case of linear function approximation, since this representation can be used without modification for the tabular case as well, given an appropriate choice of feature function ϕ, as discussed in Section 2.1.

Algorithm 1 Intrinsically motivated skill learning in MDPs.

```
 1: $t \leftarrow 0$
 2: $\theta_I \leftarrow \vec{0}$
 3: $\pi \leftarrow$ arbitrary initial policy
 4: $\mathcal{O} \leftarrow \langle U^a, M^a \rangle, \forall a \in \mathcal{A}$          ▷ Primitive action models
 5: $\vec{\rho}^a \leftarrow \vec{1}, \forall a \in \mathcal{A}$                    ▷ Initial reward models
 6: $s_t \leftarrow s \sim d(s)$                                        ▷ Initial state
 7: $B \leftarrow$ empty stack                                         ▷ Option execution stack
 8: $\mathcal{D} \leftarrow \emptyset$                                               ▷ State sample set
 9: repeat
10:     if $t$ mod $T == 0$ then                                  ▷ Planning interval
11:         $\pi \leftarrow$ plan($\{r_I\}, \mathcal{O}, \mathcal{D}$)      ▷ Approximate value iteration - Algorithm 4
12:     end if
13:     $a_t \leftarrow$ selectAction($\mathbf{s}_t, B, \pi$)          ▷ Next primitive action - Algorithm 2
14:     $\mathbf{s}_{t+1} \leftarrow$ execute($\mathbf{s}_t, a_t$)                       ▷ Observe next state
15:     update($\mathbf{s}_t, a_t, \mathbf{s}_{t+1}$)    ▷ Update models, policies, and rewards - Algorithm 3
16:     $\mathbf{s}_t \leftarrow \mathbf{s}_{t+1}$
17:     $t \leftarrow t + 1$;
18: until forever
```

An agent starts with a set of arbitrarily initialized primitive action models $\langle U^a, M^a \rangle$. These are represented and learned in the same way as universal option models, but model dynamics over only a single time step—their termination functions return 1 in all states. There are initially no temporarily abstract options in $\mathcal{O}$. Rather, the agent will create new options and augment this set over time as it reaches states that it deems important subgoals. Recall that we defer the option discovery problem—deciding which states should be considered important subgoals—to future work.

For now, we assume that the agent has a priori a set $\mathcal{G} \subset \mathcal{S}$ of states that are considered to be important subgoals. This set is partitioned into subsets $\mathcal{G}_i \subset \mathcal{G}$, each of which represents the set of states that are valid goal states for a corresponding option o_i. When an agent encounters a state $g \in \mathcal{G}_i$ for the first time, it creates an option o_i with a pseudo-reward function $\tilde{r}^{o_i}$ that is 1 for transitioning into any state $g \in \mathcal{G}_i$ and 0 otherwise. It also initializes the associated data structures needed to represent the option's policy π^{o_i} and universal models U^{o_i} and M^{o_i}. The termination function $\beta_i(s)$ for o_i is defined to be 1 if $s \in \mathcal{G}_i$ and 0 otherwise.

As described in Section 3.2, the weight vectors ρ^a, which define the intrinsic reward model for each action, are initialized to all ones. An initial state s_0 is drawn from a distribution $d(s)$ for the domain. The behavior stack B, discussed below, is initialized to an empty stack, and the state sample set $\mathcal{D}$, also discussed below, to the empty set.

Beginning with an arbitrarily initialized base policy π, every T steps, where T is a fixed interval based on computational budget, the agent computes a new base policy via truncated value iteration, as detailed in Algorithm 4, to execute for the next T steps. This planning step makes use of both primitive actions and the agent's current set of options in its backups. Once the new base policy is computed, the agent executes it for the next T steps, executing all options that are called by the policy to completion as it does so. We describe the details of the value iteration algorithm we use below, but first discuss how actions are selected from the policy it returns and what learning updates occur at each time step during the policy's execution.

Algorithm 2 Action selection.

```
1:  function SELECTACTION($\mathbf{s}_t, B, \pi$)
2:      if $B$ is empty then                              ▷ Choose option from base policy
3:          $o \leftarrow \pi(\mathbf{s}_t)$
4:          $B$.push($\langle o, \phi(\mathbf{s}_t), t \rangle$)
5:      else
6:          $\langle o, \tilde{\phi}, \tau \rangle \leftarrow B$.top()
7:      end if
8:      while $o \notin \mathcal{A}$ do                    ▷ Follow option policies until $o$ is primitive
9:          $o \leftarrow \pi^o(\mathbf{s}_t)$
10:         $B$.push($\langle o, \phi(\mathbf{s}_t), t \rangle$)
11:     end while
12:     return $o$
13: end function
```

Since options may call other options as part of their policies, the agent must maintain a behavior stack B to keep track of the currently executing options. At each time step, the agent selects a primitive action based on its current base policy π according to Algorithm 2. On a given time step, if the stack is empty then the base policy is queried with the current state. The policy may return a primitive action or an option, and this is pushed onto the stack. As long as the option at the top of the stack is not a primitive action, the policy for the option at the top of the stack is queried with the current state, and the option or action it returns is pushed onto the stack. This continues until an option policy returns a primitive action, which is then returned as the next action to execute. Note that when an option is pushed onto the stack, it is pushed as a tuple along with the feature vector for the current state $\phi(\mathbf{s})$ and the current time step t. These quantities are needed to compute the appropriate UOM learning updates when the option is terminated some number of time steps later, as described in Algorithm 3.

Algorithm 3 Update.

```
1: function UPDATE($\mathbf{s}_t, a_t, \mathbf{s}_{t+1}, B$)
2:     $\langle o, \tilde{\phi}, \tau \rangle \leftarrow B$.top()
3:     while $\beta_o(\mathbf{s}_{t+1}) == 1$ do          ▷ Update $M$ and $U$ for terminating options
4:         $\langle o, \tilde{\phi}, \tau \rangle \leftarrow B$.pop()
5:         $M^o \leftarrow M^o + \eta \left[ \gamma^\tau \phi(\mathbf{s}_{t+1}) - M^o \tilde{\phi} \right] \tilde{\phi}^T$
6:         $U^o \leftarrow U^o + \eta \left[ \phi(\mathbf{s}_t) - U^o \phi(\mathbf{s}_t) \right] \phi(\mathbf{s}_t)^T$
7:         $\langle o, \tilde{\phi}, \tau \rangle \leftarrow B$.top()
8:     end while
9:     for $\langle o, \tilde{\phi}, \tau \rangle \in B$ do          ▷ Update $U$ for continuing options
10:        $U^o \leftarrow U^o + \eta \left[ \phi(\mathbf{s}_t) + \gamma U^o \phi(\mathbf{s}_{t+1}) - U^o \phi(\mathbf{s}_t) \right] \phi(\mathbf{s}_t)^T$
11:    end for
12:    $\rho^{a_t} \leftarrow \rho^{a_t} + \zeta - \phi_i(\mathbf{s}_t)^T \rho^{a_t}$          ▷ Update intrinsic reward function
13:    if $\mathbf{s}_{t+1} \in \mathcal{G}_i, \forall \mathcal{G}_i \in \mathcal{G}$ and $o^i \notin \mathcal{O}$ then
14:        $\mathcal{O} \leftarrow \mathcal{O} \cup o^i = \langle U^{o_i}, M^{o_i}, \rho^{o_i}, \tilde{r}^{o_i}, \theta^{o_i} \rangle$          ▷ Create new options
15:    end if
16:    if $||\mathbf{s}_t - \mathbf{s}|| > \epsilon, \forall \mathbf{s} \in \mathcal{D}$ then
17:        $\mathcal{D} \leftarrow \mathcal{D} \cup \mathbf{s}_t$          ▷ Augment state samples
18:    end if
19: end function
```

The update steps of Algorithm 3 include updating the U matrices for all executing options and the M matrices for all options that terminate on the current time step using the update equations defined in Yao, et al. (2014). The weights for the intrinsic reward function are also updated at this step, based on the most recently executed primitive action. If the successor state $\mathbf{s}_{t+1}$ is found to be a subgoal state whose associated option has not yet been created, a new option for reaching that subgoal is created at this time as well, with psuedo-reward function $\tilde{r}^{o_i}$ as defined above, U and M matrices initialized arbitrarily, intrinsic reward model weights ρ^{o_i} initialized to $\vec{0}$, and parameter vector $\theta^{o_i} = \vec{0}$. The latter is the zero-initialized weight vector that will be used to approximate the option's value function and implicit policy, as discussed below.

The last step of Algorithm 3 is necessary for running Algorithm 4, a sample-based version of value iteration. Since our approach must be applicable to continuous domains, we cannot perform a full sweep of the state space required for exact value iteration. Instead, we maintain a set of state samples $\mathcal{D}$ from which we draw when performing backups. This set is initially empty, and is potentially augmented at each step based on a similarity criterion. On a given step, if the current state $\mathbf{s}_t$ is sufficiently far from all other states $\mathbf{s} \in \mathcal{D}$, $\mathbf{s}_t$ is added to $\mathcal{D}$. In our experiments we perform this distance calculation using Euclidean distance, and define the threshold for augmenting $\mathcal{D}$ in terms of a fixed parameter ϵ so that $\mathbf{s}_t$ is added to $\mathcal{D}$ if and only if $||\mathbf{s}_t - \mathbf{s}|| > \epsilon, \forall \mathbf{s} \in \mathcal{D}$. In the case of a tabular state representation, this rule amounts to adding each new state encountered to $\mathcal{D}$.

Algorithm 4 details the planning algorithm the agent executes every T steps. There are three parts to this algorithm, the first of which involves computing or updating policies for options in the agent's skill set $\mathcal{O}$. For each non-primitive option, value iteration is run for some number of iterations using the state samples in $\mathcal{D}$ that are also in the option's initiation set $\mathcal{I}^{o_i}$ to perform backups of the value function according to the option's pseudo-reward function $\tilde{r}^{o_i}$. These backups update an option's value function estimate θ^{o_i}, and define an implicit policy π^{o_i} for each option.

Algorithm 4 Planning.

```
function PLAN($\{r_I\}, \mathcal{O}, \mathcal{D}$)
    for $o_i \in \mathcal{O} - \mathcal{A}$ do                                  ▷ Update option policies.
        for $N$ iterations do
            for $\mathbf{s} \in \mathcal{D} \cup \mathcal{I}^{o_i}$ do
                $\theta^{o_i} \leftarrow \theta^{o_i} + \alpha \max_o \left[(\tilde{r}^{o_i})^T U^o \phi(\mathbf{s}) + \gamma(M^o \phi(\mathbf{s}))^T \theta^{o_i} - \phi(\mathbf{s})^T \theta^{o_i}\right] \phi(\mathbf{s})$
            end for
            $\pi^{o_i} \leftarrow \pi$ such that $\pi(\mathbf{s}) = \arg\max_o \left[(\tilde{r}^{o_i})^T U^o \phi(\mathbf{s}) + \gamma(M^o \phi(\mathbf{s}))^T \theta^{o_i}\right]$
        end for
    end for
    for $o_i \in \mathcal{O} - \mathcal{A}$ do                                  ▷ Update option models.
        for $L$ samples $\mathbf{s} \sim \mathcal{I}^{o_i}$ do
            $t \leftarrow 0$
            $S \leftarrow \langle \phi(\mathbf{s}), t \rangle$
            while $\beta^{o_i}(\phi(\mathbf{s})) \neq 1$ and $t < t_{max}$ do
                $\tilde{o} \leftarrow \pi^{o_i}(\phi(\mathbf{s}))$
                $\phi(\mathbf{s}') \leftarrow M^{\tilde{o}} \phi(\mathbf{s})$
                $U^{o_i} \leftarrow U^{o_i} + \eta \left[\phi(\mathbf{s}) + \gamma U^{o_i} \phi(\mathbf{s}') - U^{o_i} \phi(\mathbf{s})\right] \phi(\mathbf{s})^T$
                $\rho^{o_i} \leftarrow \rho^{o_i} + \left[\phi(\mathbf{s})^T \rho^{\tilde{o}} - \phi(\mathbf{s})^T \rho^{o_i}\right] \phi(\mathbf{s}^T)$
                $\phi(\mathbf{s}) \leftarrow \phi(\mathbf{s}')$
                $t \leftarrow t + 1$
                $S \leftarrow S \cup \langle \phi(\mathbf{s}), t \rangle$
            end while
            $U^{o_i} \leftarrow U^{o_i} + \eta \left[\phi(\mathbf{s}) - U^{o_i} \phi(\mathbf{s})\right] \phi(\mathbf{s})^T$
            for $\langle \tilde{\mathbf{s}}, \tilde{t} \rangle \in S$ do
                $M^{o_i} \leftarrow M^{o_i} + \eta \left[\gamma^{t - \tilde{t} - 1} \phi(\mathbf{s}) - M^{o_i} \phi(\tilde{\mathbf{s}})\right] \phi(\tilde{\mathbf{s}})^T$
            end for
        end for
    end for
    for $N$ iterations do                                                   ▷ Update exploration policy.
        for $\mathbf{s} \in \mathcal{D}$ do
            $\theta_I \leftarrow \theta_I + \alpha \max_o \left[(r_I^o)^T U^o \phi(\mathbf{s}) + \gamma(M^o \phi(\mathbf{s}))^T \theta_I - \phi(\mathbf{s})^T \theta_I\right] \phi(\mathbf{s})$
        end for
    end for
    return $\pi$ such that $\pi(\mathbf{s}) = \arg\max_o \left[(r_I^o)^T U^o \phi(\mathbf{s}) + \gamma(M^o \phi(\mathbf{s}))^T \theta_I\right]$
end function
```

The second step of the planning phase involves updating the U and M matrices to reflect the long-term effects of each option with respect to transition dynamics and expected return given a reward function. Traditionally these models would be learned from experience by executing the option's policy in the environment. This is prohibitive for two reasons. First, it requires executing options whose policies are likely to be malformed early in their learning, which may have unintended effects. Second, only a single option can be executed at a given time in this manner, and it is potentially expensive to do so depending on the domain. Rather than estimate these models from experience, therefore, we run the option policies in "simulation" using the models of existing, mature options (including primitive actions) to predict the effects of the actions selected by the policies, and use the outcome of this simulated experience to update the models.

More formally, during this phase of the planning step, for each option $o \in \mathcal{O} - \mathcal{A}$, we draw L samples of initial states from the initiation set of the option and execute the policy for the option in simulation by using the M matrices of the options it calls in its policy to determine subsequent feature vectors. This is done until the option terminates, or for some predetermined minimum number of steps t_{max}, since immature option policies may not correctly terminate. At each step of this loop, the option's U matrix is updated accordingly, and the trajectory of state visitations is recorded, along with associated time steps. Also during this loop the intrinsic reward model ρ^{o_i} for each option is updated by adjusting its weights for the corresponding visited states to be the values of the intrinsic reward model for the corresponding primitive actions in those states, as selected by π^{o_i}. When the trajectory ends, the M matrix is updated with each sample from the trajectory serving as a separate starting point, which gives us many more samples than a single update using only the start state.

The final step of Algorithm 4 computes a new exploration policy using the current intrinsic reward functions and all of the updated options. This policy is computed via N sweeps of value iteration over the agent's current state samples, and is followed for the following T steps before a new policy if computed. The choice of N in Algorithm 4 is based on computational budget, but note that the use of options will generally result in convergence of the value function much faster than using only primitive actions, and thus smaller values of N will still yield accurate approximations. This is a primary benefit of learning options and a key feature of our approach.

To recap, an agent running our algorithm begins with initially incorrect models of the effects of its actions on the environment, and begins behaving arbitrarily to gather data to improve these models. As it discovers states that are interesting in some way (e.g., because of stability of dynamics, salience, etc.), it constructs options to learn how to reliably reach these states and learns models of the long term effects of these options. The options provide a behavioral substrate on which to perform further exploration of the regions of the state space that were previously inaccessible via arbitrary exploration. Bootstrapping exploration in this way leads to construction of new options in the newly discovered regions of the state space, which allows for further exploration in areas reachable only with those new options, and so on. The following section discusses the types of domain in which we expect our algorithm to most distinguish itself as compared to more traditional exploration mechanisms.

3.4 Domains of Interest

Having adopted the MDP as the formalism for an agent's environment, it is worth taking some time to discuss the classes of MDPs for which this approach is most appropriate. Because of their generality with respect to state representation, the characteristic that best distinguishes different classes of MDPs in the absence of an extrinsic reward function (aside from simply the size of the state and action spaces) is their transition dynamics. Many simple MDPs lend themselves to efficient discovery of transition structure via random walks through the state space (i.e., an agent selecting random actions can achieve good sample coverage of the transition function). These kinds of MDPs are often not representative of real world environments, however. It is rare for random primitive action selection to yield any appreciable progress in reaching previously unreachable areas of a given state space in complex domains.

One might argue that random exploration may do better than expected in complex domains if a simulator exists whereby the agent can be started in an arbitrary state and explore from there. While this may be true, in some domains like many in robotics, the availability of such a simulator, especially one that is accurate enough to mimic real world dynamics, is often either non-existent or costly. Even if such a simulator exists, random exploration around start states that are difficult to reach through normal behavior will often lead the agent back to "easier" to reach states; i.e., failure states. Without the reliance on a simulator and the ability to sample policies using uniform initial state distributions, we would argue that most real world problems require incremental, bootstrapped learning of progressively more complex behaviors to reach new and previously unexplored areas of the state space. These kinds of domains are the motivation for the work in this thesis, and the kinds of domains on which we focus in our experiments below.

To simulate the properties of such problems in simple illustrative domains, we employ the use of *reset* mechanics in our test domains. By reset mechanics we mean that the majority of primitive actions in the majority of states reset the agent back to the environment's initial configuration. As such, random agents, or agents that use simple random action selection as an exploration mechanism, have a very low probability of reaching the majority of the states in the state space. Instead, in order to increase their ability to manipulate their environment, agents need to bootstrap their skill acquisition with their existing models and skills to purposefully navigate

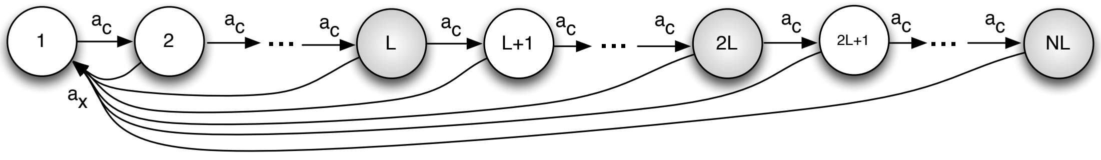

Figure 3.1. A graphical representation of the Combination Lock MDP.

through the areas of the state space with which they are already familiar. Any random action selection that may occur should occur at the fringe of the agent's expertise, and an agent's existing skill set and intrinsic reward function is precisely what keeps it at that fringe and motivated to explore there. The following section presents results validating these hypotheses in two MDPs exhibiting the characteristics discussed above.

3.5 Experiments

In this section, we present the results of experiments testing the ability of our algorithm to efficiently learn hierarchies of skills to manipulate two domains exhibiting the characteristics described in the previous section. The first is a simple but illustrative discrete MDP, and highlights the key features and behavior of the algorithm. The second is a more complex continuous MDP which affords more sophistication in the behavioral hierarchy agents can learn and also shows the applicability of our approach to continuous domains using function approximation.

3.5.1 Combination Lock

Our first experimental domain is a simple finite MDP which we call the Combination Lock. The domain is meant to be an illustrative example of how our algorithm incrementally learns an environmental model and skills of increasing complexity. As such, its simplicity is intentional and for pedagogical purposes.

The Combination Lock consists of a set of tumblers which must be turned in correct sequence in order to open the lock. An instance of the domain is parameterized by a number of tumblers, N, a tumbler sequence length, L, and a number of symbols on the lock's dial, M, the latter of which is also the number of actions ($M = |\mathcal{A}|$). These parameters determine the difficulty of reaching an arbitrary state in the domain, whose state set size is $|\mathcal{S}| = LN$.

The dynamics of the Combination Lock are as follows. At each time step, the agent turns the dial on the front of the lock to one of the M symbols on the dial. The dial turns the lowest numbered tumbler that has not yet fallen into place. Each tumbler must be turned in a unique sequence of symbols of length L before it falls into place and the next tumbler can be turned. All tumblers must fall into place in order to open the lock. As such, only a single correct primitive action sequence of length $NL - 1$ permits the opening of the lock, out of all possible M^{NL-1} sequences. Importantly, if an incorrect symbol is entered at any point in the full sequence, all tumblers are reset to their default initial position, returning the agent to the start state. All actions fail with probability p, leaving the state of the domain unchanged.

Figure 3.1 shows a graphical representation of the Combination Lock as an MDP. States are labeled from 1 to NL (left to right), with shaded states representing the successful setting of a tumbler, and unshaded states representing intermediate states. Each arrow labeled a_c indicates the action associated with the correct symbol for its corresponding state (randomly chosen for a given instance of the domain). The set of incorrect actions for each state are depicted as a single transition arc labeled as a_x to reduce clutter.

Note that the probability of an agent that executes a uniformly random policy reaching a given state s from the start state decreases exponentially in s—more precisely, when $p = 0$, this probability is $\frac{1}{M^s}$. For an instance of the Combination Lock in which $N = 20$, $L = 5$, and $M = 2$, the probability of a random agent successfully opening the lock from its initial configuration without being returned to the start state would thus be $\frac{1}{2^{99}}$, an extremely rare event.

It should be evident that any agent hoping to explore the majority of this state space when the state and action spaces are large will require deliberate exploration strategies that allow it to reliably reach high-numbered states, and to plan to do so. Our algorithm accomplishes this through the use of options and a fixed planning horizon, both of which we specify based on the value of L. In particular, for a given instance of the Combination Lock, each time an agent first visits a state whose label is divisible by L, it creates an option whose subgoal is that state, and begins learning a UOM for that option.

The expected behavior of the algorithm is initially to explore the first few states, trying different sequences of actions and building an accurate model of the dynamics around those states. This is achieved through the short planning horizon, which gives the agent a better-than-random chance of getting the first tumbler into place, at which point it will create and start learning a policy for an option to set that tumbler. As that option's model becomes more accurate, it can be used in the planning step to allow the agent to plan trajectories to states beyond that option's subgoal. This will allow the agent to explore the states leading up to the setting of the second tumbler, at which point a second option will be created. The policy for this latter option can then make use of the first option in its policy, thus decreasing the difficulty of learning the second option's policy through the use of a bootstrapped skill hierarchy. The process then continues in a similar way, with the agent learning new options with an ever-deepening hierarchy in their policies until an option is learned to open the lock from the start state.

3.5.1.1 Evaluation - Skill Learning

We tested the behavior of our algorithm on multiple instances of the Combination Lock and looked at the rate at which it discovered new areas of the state space and created options to set each tumbler. Our experiments compared the rate of discovery of new states for an agent running our algorithm with those of an agent executing a random exploration policy using only primitive actions and one executing a random policy that makes use of options learned during the planning and update steps of our algorithm. Also in this experiment we varied the number of tumblers N for the intrinsically motivated agent, keeping the sequence length L and the number of actions M constant at 5 and 4, respectively. This shows the effect of increasing the depth of the option hierarchy and the size of the state space without increasing the complexity (execution duration) of the resulting option policies.

For this experiment, we set the number of iterations of value iteration to perform at each planning step to 5, the number of state samples with which to perform option model updates to 25, the max number of simulation steps t_{max} to 25, and the planning horizon T to 20 steps. Additionally, the step size parameters in our algorithm (α, η, ζ) were all set to 0.05, and ϵ to 0.1. The failure rate p of primitive actions in the domain was set to 0.1 for all instances of the domain. Results for an average of 30 runs of this experiment are shown in Figure 3.2.

As is evident from the figure, for $N = 5$ the agent that only explores with primitive actions is incapable of reaching any but the first few states of the domain. As such, increasing the number of tumblers has no effect on its performance since it never learns to set more than the first or second tumbler, and so the runs with $N > 5$ are omitted for clarity. The agent exploring randomly with learned options does slightly better, but its discovery of new states still plateaus since its exploration is undirected. The intrinsically motivated agent, however, behaves as expected and remains at the fringe of the known areas of the state space throughout its learning, continually executing

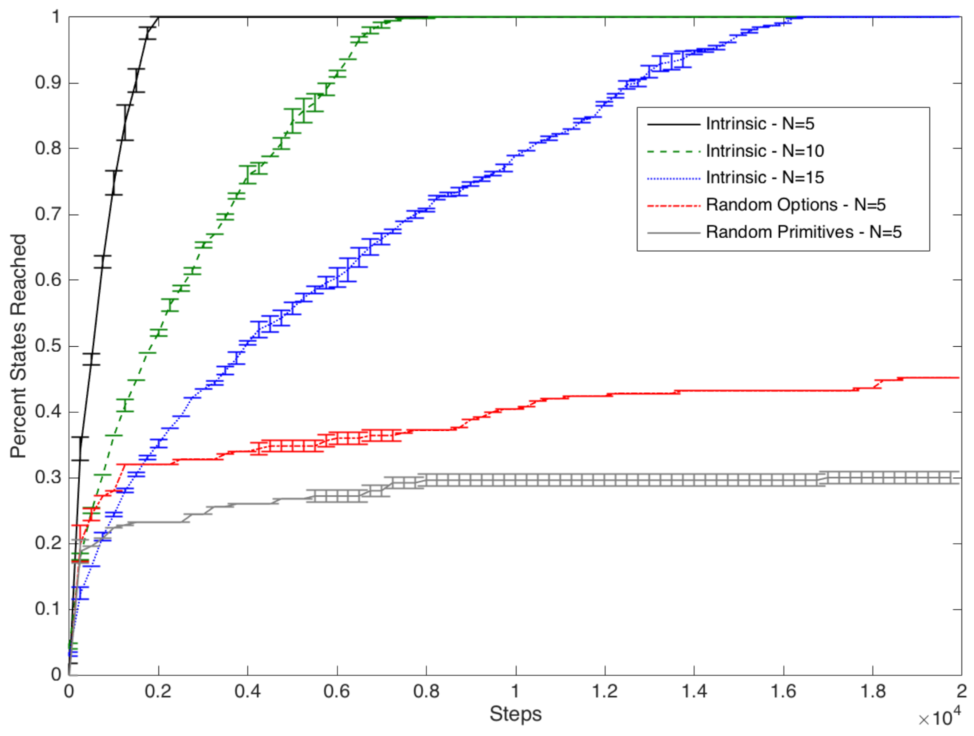

Figure 3.2. State discovery in the Combination Lock domain as a function of the number of tumblers N. Comparison with random agents for $N = 5$.

experiments to understand the dynamics of the environment at that fringe. As such, it is efficient at spending its time taking the most meaningful actions in the most meaningful areas of the state space it can currently reach.

These results held true for all values of N that we tested. As evidenced in the plot, increasing the depth of the hierarchy results in longer time to achieve full discovery of the state space, but this is only a result of the domain dynamics requiring the agent to execute more actions to reach higher numbered states. The increase in depth does not affect the agent's ability to continue to learn and discover new options. Additionally, the policies of the options learned to set each tumbler were computed correctly during the simulation steps of our algorithm for all values of N that we tested, and the agent was thus able to open the lock once it had computed the option to reach the final state.

3.5.2 Chemistry Lab

Our second domain, which we call the Chemistry Lab, affords learning a more complex behavioral hierarchy than the Combination Lock, and illustrates the applicability of our framework to domains with continuous state spaces. The domain consists of an artificial chemistry lab in which an agent can combine base elements in different ways to produce more complex compounds with varying physical properties. The lab contains a single beaker into which the agent can incrementally add small, continuously-valued amounts of base elements from a set $\mathcal{E}$ via primitive actions $a_{e \in \mathcal{E}}$. The agent also has primitive actions a_{h+} and a_{h-} to raise or lower the temperature of the beaker by a small amount via a heating element.

The state space of the domain can be modeled as an $|\mathcal{E}|+1$ dimensional space, with $|\mathcal{E}|$ dimensions corresponding to the amount of each element present in the beaker, and one dimension corresponding to the temperature of the beaker. The proportions of the elements determine the physical properties of the contents of the beaker, in particular its color. Combining different relative amounts of each element in specific orders and applying heat at the appropriate times will produce stable compounds of varying colors. The vast majority of proportions and sequences of combining them, however, produce instabilities that cause combustion events and return the agent to the lab's initial configuration (an empty beaker).

The colors of compounds in the beaker do not vary continuously, but rather undergo discrete phase shifts, immediately changing from red to blue, for example. These shifts in color occur in small regions of the state space and are used to define subgoals in the domain. For example, a red compound might retain its properties in response to applying heat for some time, but change to green after hitting a certain threshold temperature. An agent running our algorithm in this domain will learn options to create these stable compounds of varying colors as it discovers them through exploration.

In our experiments, we used an element set $\mathcal{E}$ of size 2. Each element can be present in the beaker in an amount that varies continuously from 0 to 1, and has an associated action that adds a small amount of that element to the beaker. The amount of an element added by its associated action is determined by drawing from a Gaussian distribution with mean 0.05 and standard deviation of 0.01. This provides some stochasticity to the agent's actions. Units of elements can only be added to the mixture, not removed. If the amount of an element present in the beaker ever becomes higher than 1, this creates an instability, resulting in a combustion event and the beaker returning to the empty state.

The heating element can be controlled via two actions, one to raise the temperature and one to lower it. The resulting change in temperature when executing these actions is also drawn from Gaussian distributions of mean 0.05 and -0.05, respectively, and 0.01 standard deviation. The temperature of the beaker varies from 0 to 1 as well, and is set to 0.5 in the domain's initial configuration. Attempting to lower the temperature below 0 or raise it above 1 has no effect.

The amount of each element present in the beaker determines the color of the compound in the beaker, with each element and the termperature corresponding to one of the red, blue, or green components of the mixture's color. As mentioned above, however, the color does not change continuously as more of an element is added. Rather, there are regions of the state space (hyperspheres of radius 0.02) which, when entered, result in the beaker's contents changing to the color associated with the center point of the hypersphere. These hyperspheres define the goal regions $\mathcal{G}_i \subset \mathcal{S}$ of the agent's options.

The dynamics of the domain are such that creating certain compounds is a necessary step to creating others, because the regions of stability of the former lie on the path through the state space required to create the latter. Furthermore, some of these intermediate compounds are precursors to several different successor compounds. This property distinguishes the Chemistry Lab from the Combination Lock in that the graph of option dependencies is more tree-like, rather than a simple chain. There are 27 different stable compounds of distinct color in the instance of the domain we used in our experiments, and thus 27 options for an agent to learn.

3.5.2.1 Evaluation - Skill Learning

We tested an agent's ability to learn a complete set of options in the instance of the Chemistry Lab—one for each of the 27 subgoals $\mathcal{G} \subset \mathcal{S}$ as described above. We used a set of 1,000 Gaussian radial basis functions (RBFs) evenly spaced over the 3-dimensional state space as our function approximator. The activations of these basis functions are given by

$$\phi_i(\mathbf{s}_t) = e^{-\beta||\mathbf{s}_t - \mu_i||}, \tag{3.3}$$

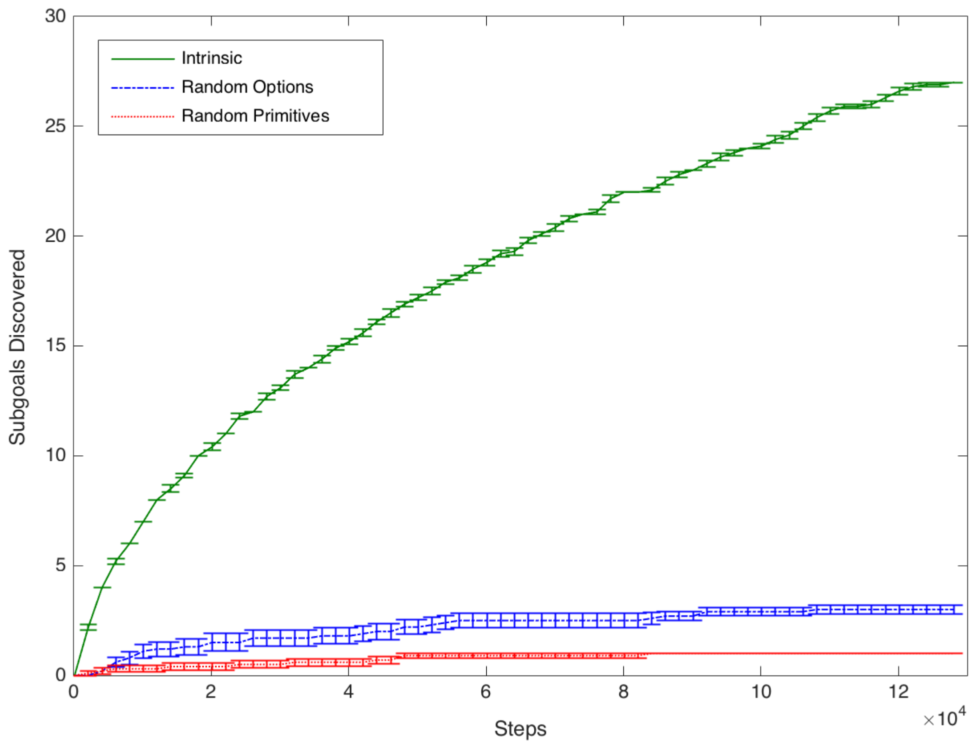

Figure 3.3. Subgoal discovery in the Chemistry Lab domain.

where $\mathbf{s}_t$ is the state vector at time t, μ_i is the vector of equal cardinality corresponding to the center of the i^{th} basis function, and β is a parameter determining the width of each kernel, which was set to 80 in our experiments.

For this experiment, we set the number of iterations of value iteration to perform at each planning step to 5, the number of state samples with which to perform option model updates to 50, the max number of simulation steps t_{max} to 50, and the planning horizon T to 30 steps. Additionally, the step size parameters in our algorithm (α, η, ζ) were all set to 0.02, and ϵ to 0.05. Results for an average of 30 runs of this experiment are shown in Figure 3.3.

As in the Combination Lock domain, the agent running our algorithm learns to stay at the forefront of its capabilities in terms of predicting the effects of its actions and its ability to manipulate its environment, maximizing the speed at which is discovers new subgoals and learns options to reach them. As these new options are learned, they are themselves used as part of the exploration policy to venture further into the state space and discover new subgoals, and the process repeats.

We again compared our agents with random agents choosing either from primitive-only actions sets or actions sets with primitives and learned options, the latter of which were computed with the same machinery as the intrinsically motivated agents once their corresponding subgoals were discovered. Again, random exploration is only able to achieve a certain base level of expertise, after which the inability of those agents to focus their efforts on the highest information areas of the state space results in plateaus in their learning process.

3.5.2.2 Evaluation - Task Performance

Learning skills during a developmental period is of little use if the skills are not applicable to the solution of novel problems later in life. To evaluate the utility of the agent's learned skills, we tested the agent at regular intervals during its developmental phase by generating reward functions that define tasks to create compounds with randomly selected colors. These reward functions are each passed to the agent, which then runs value iteration to compute a policy to make the corresponding compounds. The policies are executed by the agent and its total reward over the testing interval is recorded. Rewards for these tasks are 1 for transitioning into states in which the beaker contains a compound with the target color, and 0 otherwise.

As the agent continues to refine its models and learn new skills, its ability to create a wider array of compounds should increase, and its total reward during the testing phases should increase with each evaluation as well. In our experiments we ran this testing phase every 1,000 steps, testing for 500 steps each time. For each testing phase we randomly chose compounds from the set of stable compounds that exhibit

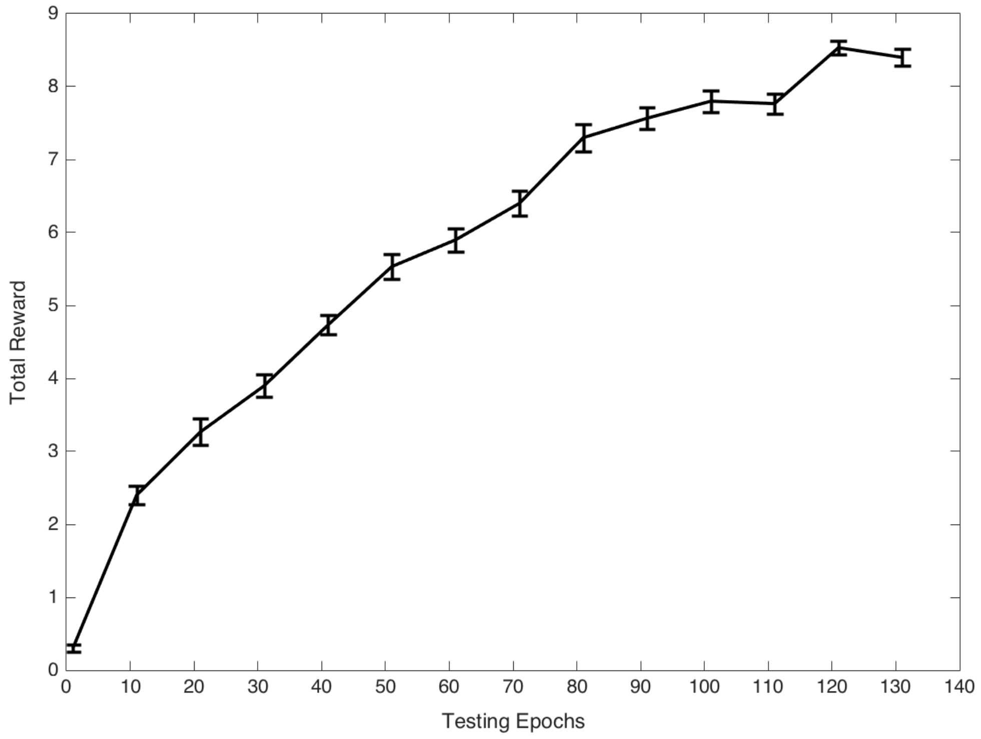

Figure 3.4. Extrinsically rewarded task performance in the Chemistry Lab domain.

distinct properties—the same set from which the agent's subgoals were determined. The number of primitive actions required to solve each task optimally varied from approximately 10 steps to approximately 90 steps, depending on the location of the goal.

Figure 3.4 shows the performance of our agents on these randomly selected tasks as a function of testing epoch. The plot is an average of 30 runs and error bars represent standard error. As is evident, the performance of the agent increases with each testing epoch as the agent discovers new subgoals and learns options to reach them in the exploration phases between each test epoch. This continues until the agent has learned options to reach all subgoals, after which it behaves optimally for extrinsically rewarding tasks drawn from this set of subgoals. This illustrates the accuracy of the learned option policies and their utility in solving extrinsically rewarding tasks which had never been extrinsically rewarding previously in the agent's lifetime.

3.6 Summary and Future Work

We have presented an algorithm for intrinsically motivated skill learning in MDPs and shown that it can be used to discover complex behavioral hierarchies in both discrete and continuous domains. The algorithm learns a skill hierarchy based on pre-selected subgoals pertinent to the domain, which it can then use to efficiently compute policies to solve novel tasks in the domain that it never explicitly encountered during its developmental period. The learning of this hierarchy is bootstrapped, always using the agent's current skill set and models to explore fringe areas of the state space where its model is inaccurate, and learning skills to navigate those areas of the space as the model becomes more accurate. We showed in our results for the Chemistry Lab experiments that this hierarchy can be used to efficiently compute policies for unfamiliar tasks after some developmental period, and that the time and computation expended to learn the hierarchy is less than that of learning those tasks individually.

As mentioned above, one of the limitations of our algorithm is the lack of automatic selection of subgoals, which is an important open problem in hierarchical RL. While there have been several approaches to solve this problem in a domain-independent way (Simsek, 2008; Hart et al., 2008; Konidaris, 2011), it is likely that there is not a single universal method for selecting subgoals. Rather multiple different selection criteria, some domain-dependent, should be used to construct efficient and useful skill hierarchies in a given domain. Experimenting with these different selection criteria is an interesting avenue for future research.

Another interesting line of experimentation involves the form of intrinsic reward function we chose to guide exploration behavior. There are many possible alternatives to using model confidence as the basis for intrinsic reward, and it is interesting to think about what other functions may improve the speed of model learning and skill acquisition. Certain forms of intrinsic rewards could also potentially be used to address the issue of subgoal selection mentioned above. It is also possible to search directly in reward function space for functions that maximize agent performance averaged over lifetime, as Lewis, et al. (2010) show. Certainly multiple intrinsic reward functions may be combined in various ways and determining appropriate ways to combine them is also of interest.

Finally, the state representation we used in this chapter was in terms of feature vectors over a monolithic state space as defined by the MDP formalism. This poses some issues relating to the curse of dimensionality, namely that adding any information to the state representation (e.g., a single bit in the case of a finite MDP, or another dimension in the case of a continuous MDP) results in an exponential increase in the number of features needed to approximate functions over the state space with the same accuracy. Choosing features for a function approximator that generalize well can help to alleviate this problem and greatly reduce both the size of the representation and the sample complexity for approximating functions over it. One technique for achieving these improvements can be used when the state can be represented as a vector of values of distinct variables, each of which only depend on a small subset of the others when determining state transitions and rewards. We discuss this technique and an algorithm for applying it to intrinsically motivated exploration in the following chapter.

CHAPTER 4

INTRINSICALLY MOTIVATED SKILL LEARNING IN STRUCTURED ENVIRONMENTS

In the previous chapter, we presented an algorithm for intrinsically motivated learning of skill hierarchies in MDPs. A limitation of this approach is its susceptibility to the curse of dimensionality, since every variable that defines the domain adds another dimension to the state space. One way to alleviate this is to exploit the structure of environments in which the values of certain variables are irrelevant to predicting the values of others. Exploiting this structure can allow functions over the state space to be approximated in many fewer dimensions than the dimensionality of the full state space. In this chapter, we present an extension of the principles we developed in the previous chapter to environments in which this structure can be exploited via state abstraction. The formalism we use for such environments is the factored MDP (FMDP), described in Chapter 2.

Recall from Chapter 2 the work of Jonsson and Barto (2006), who propose to learn the full structure of an FMDP with a fixed reward function and then use the VISA algorithm to decompose the task into sub-problems solved by exit options. By contrast, the approach we develop here concerns the scenario in which there is (at least initially) no single problem the agent must solve, and consequently no extrinsic reward function. An agent in this scenario accumulates structural knowledge as it explores its environment, and creates skills (options) for reaching different abstract subgoals as enough structure becomes available to do so. For many subgoals, the feasibility of learning an option to reach them will occur long before the full structure of the environment is discovered. Indeed, the point of this approach is that incrementally constructing options before the full structure is discovered will increase the probability of an agent being able to reach areas of the state space that would otherwise be quite difficult to reach, thereby enabling the agent to learn about the structural properties of those areas. The following sections detail the components of an algorithm for achieving this behavior and present the results of its behavior in a non-trivial FMDP.

4.1 Incremental Structure Learning

An agent employing the algorithm we define in this chapter must have some mechanism by which it can iteratively and incrementally refine the representation of the conditional probability distributions that define an FMDP's transition function. For this purpose, we use a modified version of the approach given in Jonsson and Barto (2007), described in some detail in Section 2.5. The specific method of structure learning is not a critical aspect of this framework, however. Other methods may be used to learn the conditional probability trees (CPTs) of the DBN model as long as they are incremental in nature and not computationally prohibitive.

The modification to the approach of Jonsson and Barto (2007) we make is to add a mechanism for deleting refinements when necessary. Recall from Section 2.5 that their method greedily adds edges to the DBN models of the environment according to the BIC metric in an incremental fashion. We found that occasionally an incorrect refinement is made. To remedy this, at each time step a χ^2 test of significance is performed on the distributions over the domain values of the variable at each non-leaf node of each CPT, with and without the current refinement, to see that their difference is significant. If this significance drops below a certain value (0.995 in all of our experiments) the refinement is removed by pruning the tree at that node and placing all of the data from that subtree into the newly formed leaf node. In our experiments, this remedied the few occasions in which an incorrect refinement was made and allowed the agent to recover from a mistake that would otherwise have precluded further correct structure learning.

4.2 Caching Options

The framework outlined in Jonsson and Barto (2006) proposes to learn the full structure of an FMDP given a specified reward function and only then to use the VISA algorithm to decompose the task into sub-problems solved by exit options. To apply these techniques to an intrinsically motivated exploration scheme, where there is no specified extrinsic task, an agent should be able to accumulate structural knowledge as it explores its environment and create options for reaching various subgoals as enough structure becomes available to do so. For many options, this will occur long before the full structure of the environment is discovered. Indeed the point of this approach is that incrementally constructing options before the full structure is discovered will increase the probability of an agent being able to reach areas of the state space that would otherwise be quite difficult to reach, thereby enabling the agent to learn about the structural properties of those areas.

In order to cache options in this way, an agent must monitor changes in the structure of its model, and each time the structure is changed evaluate the resulting model to decide whether a new option may be constructed. To do this, the agent maintains a set $\mathcal{C}$ (initially empty) of what we term *controllable* variables. A controllable variable is one for which the agent possesses a set of options capable of setting the variable to any of its possible values. Every time a new refinement of a leaf in the CPT for variable S_i in the DBN for action a is made, if $S_i \notin \mathcal{C}$ the causal graph of the domain is checked to see if each of its ancestors is controllable. This is to make sure that the agent can reliably reach the context given by the branch along which the new refinement has been made. If this is true, and the value of S_i is possibly changed by executing a in the branch's context, an exit option (and its associated transition and reward models) is created to reach that context and execute action a. If the new option, coupled with all existing options, results in the agent's ability to set S_i to each of its possible values, S_i is added to $\mathcal{C}$.

As in Jonsson and Barto (2006), the reward function that specifies the subtask an option is created to solve (known as a pseudo-reward function) is -1 for every state in which the option's exit context is not satisfied, and 0 for states in which the context is satisfied. Since the pseudo-reward functions for the options created in this way are known, the SVI algorithm can be used to compute their policies, as distinguished from Jonsson and Barto (2006), in which unstructured RL algorithms were used to learn the option policies from experience. Additionally, the SVI algorithm has at its disposal the agent's current set of options (and their corresponding models) for setting each variable in $\mathcal{C}$ to each of its values, which will in general lead to faster computation of new option policies for the reasons described in Chapter 4. This computation requires that the transition and reward models for each option be computed as they are constructed as well, which is done using the algorithm given in Jonsson and Barto (2006). In contrast to Jonsson and Barto (2006), however, in which only primitive actions were used in option policies, the options constructed by an agent in this framework may contain recursive calls to other options in the agent's skill set.

There is one other issue which must be considered when deciding whether to construct an option in this framework—an issue that is a direct consequence of bootstrapping the learning of new skills based on recently learned skill and models. It may be the case that a refinement is made in the CPT for S_i, and all ancestors of S_i are controllable, but the CPT is either incomplete or incorrect in some way. If the agent were to create an option at this point, it would likely be incorrect (both its policy and its model). Thus we need a way to decide whether the correct CPT has been learned for S_i under action a. If the environment is deterministic, then once the entropy of the distribution at every leaf of the CPT has reached zero, no more refinements can be made and we know the correct structure has been discovered. This is a strong assumption, however, and applies only to less interesting domains.

When the environment is stochastic, our choice of structure-learning algorithm prevents us from being able to distinguish incomplete structural knowledge from inherent stochasticity, since greedy methods like it are not always guaranteed to find the correct structure. We discuss this disadvantage in Section 4.6. Rather than attempting to make this distinction, however, agents in our framework construct options in the absence of knowledge about structural correctness, and instead monitor the utility of their current set of options, abandoning those whose empirical success rates do not match their expected success rates.

More formally, each time the structure of a DBN is modified by the structure-learning algorithm, if the latest refinement results in a context-action pair that alters the value of some variable, an option to set that variable to the new value is created and added to the agent's set of options, $\mathcal{O}$. Each option in $\mathcal{O}$ is assigned a success rate, σ, initially equal to the expected success rate, σ^*, of the option. The expected success rate is obtained from the leaf of the CPT corresponding to the option's exit context in the DBN corresponding to the option's exit action, and is equal to the probability that the variable the option is intended to change will take on its intended value when the exit action is executed in the exit context. Every time an option is executed, it is allowed to run to completion for a maximum of M time steps. If within that number of steps the option's context is reached, its exit action is executed, and its objective is achieved (i.e., its associated variable is set to its desired value), then the execution is considered successful. Otherwise, the execution is considered unsuccessful.

After the k^{th} execution of option o, o's success rate, σ_o, is updated according to

$$\sigma_o \leftarrow \sigma_o + \frac{1}{k}(\delta - \sigma_o), \tag{4.1}$$

where δ is 1 if the option was successful, and 0 otherwise, so that σ_o always reflects the average empirical success rate of o. If at any time after at least N executions of o, σ_o drops below o's expected success rate, σ_o^*, by more than a factor η, the option is removed from $\mathcal{O}$, along with any options that reference o in their policies. The agent then continues to explore with its remaining skill set, the process of discovering new structure, constructing options, and testing their utility continuing until all variables are controllable. Should the agent reach that point, it then has a set of options it can use to efficiently compute a recursively optimal solution to a wide array of potential tasks in its domain via the SVI algorithm. With this machinery in place for incrementally adding options as enough structure becomes available to do so, we next describe our method for employing intrinsic motivation to maximize the rate at which DBN structure is learned.

4.3 Intrinsically Motivated Structure Learning

The discovery and construction of options in our framework is obviously dependent upon the efficient discovery of dynamical structure in an agent's environment—structure that informs the agent how certain features of its world are influenced by its actions. It is therefore essential to our approach that an agent be able to discover this structure in a timely manner through its own experience. Random action selection in complex environments is very unlikely to produce efficient structure learning, since often complex sequences of actions must be performed just to allow the agent to reach certain areas of the state space, let alone collect sufficient statistics about them from repeated trajectories. In fact, as we will show, even the active structure learning technique presented in Jonsson and Barto (2007) will often fail to discover much of the structure of such environments.

For this reason we propose the use of intrinsic reward to guide an agent to areas of the state space for which its dynamical model is lacking in accuracy, thus allowing it to collect relevant information about those areas and improve its model. The improved model may then afford the construction of new options to manipulate features of the environment in those areas, which may in turn provide access to new areas that require further exploration. This use of intrinsic motivation to improve model quality has been proposed in various forms before (Schmidhuber, 1991; Kaplan & Oudeyer, 2004). These approaches, however, are seemingly concerned with model improvement for the sake of model improvement. It is our contention that, while models are an important component of intelligent systems, their construction should not be the primary objective. Indeed the true goal for an intelligent system should be improved control of its environment; that is, the improvement of a world model should be in service of learning skills that allow an agent to control its environment, though the learning of those skills may be informed by a world model, as in the approach presented here.

We build upon the approach to active structure learning in FMDPs proposed by Jonsson and Barto (2007) and introduce a novel extension that allows for discovery of structure in more complex environments. In our scheme, an agent uses its current skill set to perform "experiments" in its environment so as to expedite structure learning. An experiment, like an exit, is composed of a context and an associated primitive action. Similar to Jonsson and Barto (2007), we seek to find the best experiments to perform by calculating potential changes in distribution vector entropies at CPT leaves and picking the experiment that results in the largest change. Rather than simply looking at leaves whose contexts are satisfied by the current state, however, we can also consider leaves whose contexts consist exclusively of controllable variables, since the agent possesses options to reliably set those variables to any of their values. Additionally, we can check to see which settings of the controllable variables that are not part of the leaf's context will yield the highest gain at that leaf.

We thus consider the context of the best experiment to be a setting of all controllable variables that maximizes this gain. When the experiment with the largest gain is found, we create an intrinsic reward function that is 1 if the experiment's context is satisfied, and -1 otherwise. With this reward function, we compute a policy using SVI to reach that context and execute the action associated with the leaf's CPT. This search and policy computation is carried out with probability $1 - \epsilon$ each time a desired context is reached and the appropriate primitive action is executed, where $\epsilon \in [0, 1)$. Otherwise a random action from the agent's action set (including options) is selected and executed to completion.

We can always do this because we only consider leaves whose contexts are controllable. Since the agent starts out with no controllable variables, initial exploration is carried out according to the local active learning scheme in Jonsson and Barto (2007) in which only leaves whose contexts are satisfied by the current state are considered when choosing actions. As enough structure is discovered and certain variables become controllable via construction of low-level options as outlined in the previous section, however, the agent can use those new skills to reliably set contexts for which it has limited or uneven samples at the leaves of its CPTs. In this way, structure learning is bootstrapped, using existing structural and procedural knowledge. For domains with hierarchical structure in which it is not necessary to know the full structure of the domain to compute lower level skills this approach should offer a distinct advantage over active exploration schemes that use only local information to choose actions. We show this to be the case in Section 4.5, after giving the full details of our algorithm in the following section.

4.4 Algorithm

Algorithms 5, 6, and 7 give the full details of our algorithm for intrinsically motivated skill learning in FMDPs. Algorithm 5 shows the main loop of the algorithm, which performs initialization of the CPTs and makes calls to the **selectAction** (Algorithm 6) and **update** (Algorithm 7) functions at each time step.

Algorithm 5 Intrinsically motivated skill learning in FMDPs.

```
1: Initialize CPTs to single-leaf trees (no refinements)
2: $\mathcal{O} \leftarrow \emptyset$
3: $t \leftarrow 1$
4: $s^t \leftarrow$ initial state
5: repeat
6:     $a^t \leftarrow$ selectAction($\mathbf{s}^t$)                          ▷ Algorithm 6.
7:     Execute primitive action $a^t$ and observe $\mathbf{s}^{t+1}$
8:     update($\mathbf{s}^t, a^t, \mathbf{s}^{t+1}$)                         ▷ Algorithm 7.
9:     $\mathbf{s}^t \leftarrow \mathbf{s}^{t+1}$
10:    $t \leftarrow t + 1$
11: until forever
```

The **selectAction** function performs the task of selecting a primitive action at each time step based on the currently executing options, if any. Since options may call other options, this function may need to recursively query option policies to find a primitive action to return. The function also handles monitoring for immature options, terminating any option that has been running for more than M steps and updating the option's σ value accordingly. If an option is deemed too immature ($\sigma_o < \sigma_o^* - \eta$), then it is removed from the option set along with any of its descendants, and their associated variables removed from the set $\mathcal{C}$ if necessary.

If there are no currently executing options, then a policy to perform the currently most informative experiment is computed with the SVI algorithm using a reward function calculated based on the context-action pair that would yield the highest information gain, as described above. The first primitive action of this policy (which may involve nested options) is then returned.

The **update** function handles updating the agent's data structures at each time step based on the transitions it observes when taking actions. It first checks to see whether any of the currently running options has terminated in the newly observed

Algorithm 6 Select next primitive action.

```
function selectAction($\mathbf{s}^t$)
    if any option $o$ is currently executing then
        if $o$ has been running for less than $M$ steps then
            return next primitive action of $o$'s policy (may involve nested options)
        else
            Terminate $o$
            Update $\sigma_o$ using Equation 4.1 (with $\delta = 0$)
            if $\sigma_o < \sigma_o^* - \eta$ and $o$ has been executed at least $N$ times then
                $\mathcal{O}^- \leftarrow o \cup$ descendants of $o$
                $\mathcal{O} \leftarrow \mathcal{O} \setminus \mathcal{O}^-$
                for each option $n \in \mathcal{O}^-$ do
                    if $n$'s exit variable $S_i \in \mathcal{C}$ then
                        $\mathcal{C} \leftarrow \mathcal{C} \setminus S_i$
                    end if
                end for
            end if
            return selectAction($\mathbf{s}^t$)
        end if
    else if $\mathcal{O} = \emptyset$ then
        return best primitive action given by method in Jonsson and Barto (2007)
    else
        Compute policy $\pi$ to reach best context given $\mathcal{O}$ using SVI algorithm
        return first primitive action of $\pi(\mathbf{s}^t)$ (may involve nested options)
    end if
end function
```

Algorithm 7 Update data structures.

```
function UPDATE($\mathbf{s}^t, a^t, \mathbf{s}^{t+1}$)
    if any currently executing options $o$ are terminal in $\mathbf{s}^{t+1}$ then
        Update $\sigma_o$ using Equation 4.1 (with $\delta$ based on success or failure)
    end if
    for each $S_i \in \mathbf{S}$ do
        Add sample $\langle \mathbf{s}^t, f_{S_i}(\mathbf{s}^{t+1}) \rangle$ to appropriate leaf of tree for $S_i$ in DBN for $a_t$
        Compute BIC scores for each potential refinement at leaf (Equation 2.13)
        if refinement is made at leaf then
            Distribute samples to appropriate children
            if new context causes change in value of $S_i$ then
                Create exit option(s) $\mathcal{O}_{new}$ to set $S_i$ to new values
                $\mathcal{O} \leftarrow \mathcal{O} \cup \mathcal{O}_{new}$
                if $\mathcal{O}$ contains options to set $S_i$ to all possible values then
                    $\mathcal{C} \leftarrow \mathcal{C} \cup S_i$
                end if
            end if
        end if
        for each internal node in context given by $\mathbf{s}^t$ do
            if refinement is no longer significant then
                Prune tree at node and collect all samples into new leaf
                Delete options and any descendants associated with former context
                if new context causes change in value of $S_i$ then
                    Create option(s) to set $S_i$ to appropriate value(s)
                end if
                Update $\mathcal{O}$ and $\mathcal{C}$ accordingly
            end if
        end for
    end for
end function
```

state and updates those options' σ values accordingly based on whether they resulted in the expected outcome or not.

Next it processes the sample consisting of the previous state and the newly observed state, adding the sample to the appropriate leaf of the CPT associated with the action taken. The BIC scores for the potential refinements at that leaf are then computed to determine whether a refinement should be made. If the scores suggest that a refinement should be made, then leaf is split on the values of the refining variable and the samples at the leaf are distributed to the new child leaves appropriately. If the refinement causes a change in the value of a variable, then one or more options are created to set the value of that variable to each of its possible values, and those options added to the agent's option set.

The update function also handles checking for incorrect refinements, performing a χ^2 test of significance at each internal node of the context for $\mathbf{s}^t$ and pruning the tree at that node if the test fails. If the tree is pruned, the samples from the entire subtree are pooled into the newly formed leaf and any options associated with contexts in the former subtree are removed from the agent's option set. If the context created by the pruning yields a change in the value of a variable then new options are created if necessary to set that variable to each of its values.

This algorithm is run indefinitely until the agent possesses options to set the values of each of the state variables to any of their possible values, at which point the agent can use its option set to compute a policy to reach any state in the environment. The following section discusses some experiments we performed testing the validity of our algorithm on a large structured domain, and shows the utility of the algorithm in solving multiple tasks in the environment after it has learned a useful set of options.

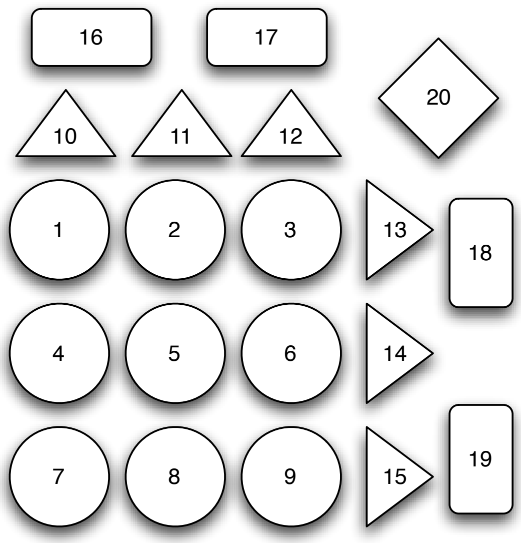

Figure 4.1. A visual rendering of the Light Box domain.

4.5 The Light Box Domain

4.5.1 Dynamics

We conducted experiments in a simple but large artificial domain called the Light Box (Figure 4.1). The domain consists of a set of twenty "lights", each of which is a binary variable with a corresponding action that toggles the light on or off. Thus, there are twenty actions, $2^{20} \approx 1$ million states, and approximately 20 million state-action pairs. The nine circular lights are simple toggle lights that can be turned on or off by executing their corresponding action. The triangular lights are toggled similarly, but only if certain configurations of circular lights are active, with each triangular light having a different set of dependencies. Similarly, the rectangular lights depend on certain configurations of triangular lights being active, and the diamond-shaped light depends on configurations of the rectangular lights.

In this sense, there is a strict hierarchy of dependencies in the structure of this domain. Figure 4.2 shows the causal graph of the instance of the Light Box domain we used in our experiments, illustrating the dependencies between each of the variables.

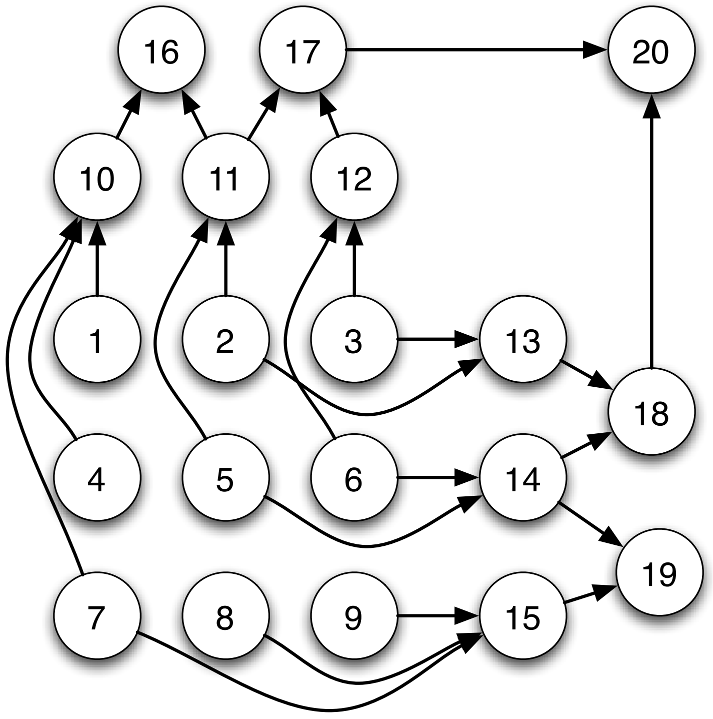

Figure 4.2. The causal graph of the Light Box domain.

To remove clutter, the reflexive dependencies are not drawn, but each light obviously depends on its own value at the previous time step. With the exception of reflexive dependencies, each link in the causal graph indicates that the parent light must be "on" in order to satisfy the dependency. Each action has a 0.9 probability of toggling its associated light as long as the light's dependencies are satisfied, and a 0.1 probability of leaving the light unchanged. If an action is taken to toggle a light whose dependencies are not currently satisfied, however, the entire domain is reset to all lights being off.

The domain was designed to emulate scenarios in which accurate lower-level procedural knowledge is essential for successful learning of more complex behaviors and their environmental effects. Because of the "reset" dynamics, random action selection is extremely unlikely to successfully turn on any of the lights at the top of the hierarchy. Additionally, structure-learning is quite difficult using only local active learning schemes. An agent must learn and make use of low-level skills in order to be able to remain in the more difficult-to-reach areas of the state space in which it can learn higher-level skills. We also emphasize that the agent does not perceive any structure directly as may be evident in the visual rendering of the domain. Rather the agent perceives only a string of twenty bits as its state. The structure must be discovered from the state transitions the agent experiences while interacting with its environment, and thus the discovery of hierarchy is non-trivial.

The options that are discovered in the Light Box domain may have nested policies, the relationship between two of which is shown in Figure 4.3. The policies are represented as trees, with internal nodes representing state variables and leaves representing action choices, which may be either primitive actions or options. Branches are labeled with the possible values of their parent variables. In the example shown, the policy for the option to turn on light number 16 (O_{16}) contains at one of its leaves another option (O_{10}) to turn on light number 10, which is one of the dependencies for

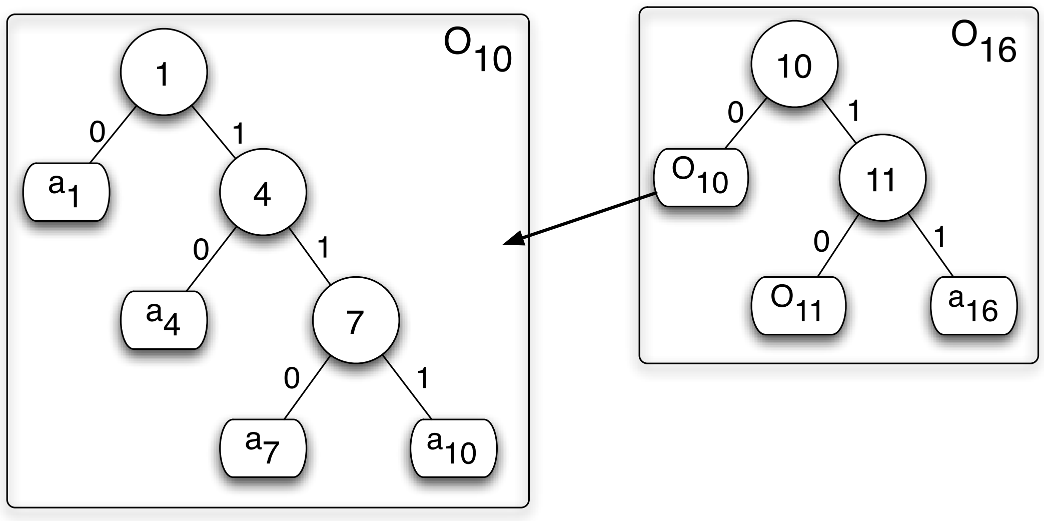

Figure 4.3. Examples of compact option policies in the Light Box domain. Internal nodes represent state variables, leaves represent action (option) choices. Branches are labeled with state variable values. Notice the nested policies.

light number 16. This nesting of policies is a direct result of the hierarchical nature of the domain.

4.5.2 Structure-Learning

To evaluate our proposed scheme for active structure learning we compared the performance of agents using three different types of exploration policies to guide behavior while learning the structure of the Light Box domain. All agents executed the same structure-learning algorithm discussed above and incrementally created options according to the scheme described in the previous section, also deleting options based on their empirical success rate when appropriate. The number of samples, k, required to make a refinement at a leaf of a CPT was 20. The maximum number of steps, M, that options were allowed to execute was 50. The minimum number of executions, N, needed to evaluate the utility of an option was 20. And the maximum discrepancy,

η, allowed between the empirical success rate, σ, and the expected success rate of an option, σ^*, was 0.1.

The Random agent selected a random action from the agent's set of actions (including options) and executed each one to completion before choosing another. The Local agent employed the active learning scheme presented in Jonsson and Barto (2007), except that when a random action was taken, the action was chosen randomly from the agent's entire set of available actions (including options) and executed to completion. The exploration parameter ϵ for the Local agent was set to 0.1. The Global agent employed our intrinsically motivated active learning scheme, which uses more global information when selecting actions and computes plans to reach more informative areas of the state space. The choices for parameter values in each agent were made via a rough search of parameter space and based on reported values in previous work when applicable. We did not notice much sensitivity in performance as a result of changing these values slightly, though of course the parameter M will in general be largely dependent on the domain, which is a limitation of this technique.

Since we had access to the true transition structure of the instance of the Light Box, we could compare the refinements made by each agent at a given time step to the set of refinements that define the correct model and plot the accuracy of the model for each agent over time. Figure 4.4 shows the number of correct refinements discovered by each agent as a function of the number of time steps. The learning curves presented are averages of 30 runs for each agent. Error bars show standard deviation. Clearly the hierarchical nature of the domain makes structure learning very difficult for agents that cannot plan ahead in order to reach more informative areas of the state space. Both the Random and Local agents are able to learn what is essentially the bottom layer of the hierarchy, but once this structure is discovered they continually sample the same areas of the state space and further learning is stalled.

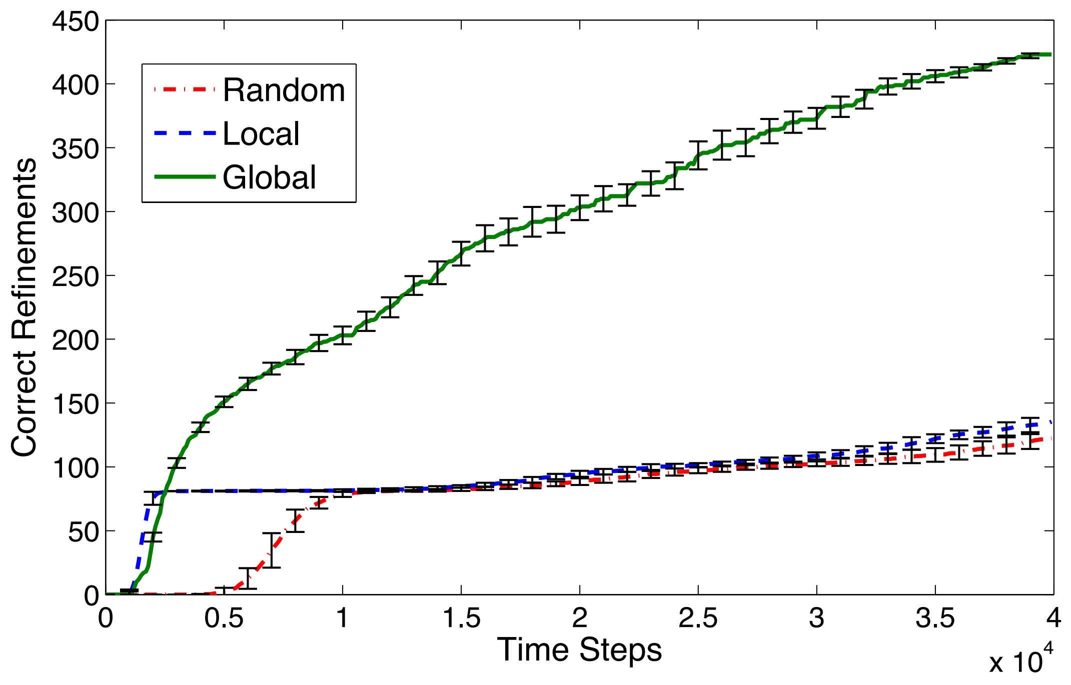

Figure 4.4. Structure-learning performance for three different exploration policies.

The Global agent on the other hand uses the options constructed from this initial structure to perform useful experiments in its environment, allowing it to reach areas of the state space that the other agents cannot reach reliably, and thus uncover more of the domain structure. This structure is then used to generate new skills that enable further exploration not possible with only the previous set of skills. This bootstrapping process continues until all of the domain structure has been discovered, at which point the agent possesses options to set each light to either on or off. There are 423 refinements in the true DBN model of this instance of the Light Box, all of which the Global agent was able to find in each run. Note also that exploiting the structured representation of the environment allows the agent to uncover the transition dynamics without even visiting a vast majority of the states in the domain, with the Global agent finding the correct structure in under 40,000 time steps reliably.

4.5.3 An Ensemble of Tasks

We also conducted experiments to illustrate the utility of computing hierarchies of skills for ensembles of tasks in large factored domains such as the Light Box. We compared the time it took to compute policies using the SVI algorithm for various tasks (i.e., different reward functions) for an agent with only primitive actions to the time taken by one with a full hierarchy of options (including primitives). For each of the twenty lights we computed a policy for a task whose reward function was 1 when that light was on and -1 otherwise. We averaged together the computation times of the tasks at each level of the Light Box hierarchy (i.e., all times for circular lights were averaged together, and similarly for triangular and rectangular lights, with only one task for the diamond light). Experiments were run using unoptimized Java code on an Intel 2.4GHz quad core processor with 4GB of RAM. The time spent computing option policies and corresponding models for the agent with options was 21.76 seconds.

Results are shown in Figure 4.5. For the lowest level of the hierarchy, where the tasks can be solved by one primitive action, the two agents take very little time to compute policies, with the options agent being slightly slower due to having a larger action set through which to search. Once the tasks require longer sequences of actions to solve, however, we see a significant increase in the computation time for the primitives-only agent, and little or no increase for the options agent. The overhead of computing the options in the first place is thus compensated for once the agent has been confronted with just a few different higher-level tasks. The savings become very substantial above level 2 (note the log scale). Of course the complexity of this domain can be increased by increasing the number of dependencies in its structure, but our results show that for even as few as two or three dependencies per variable the benefits of computing options are drastic.

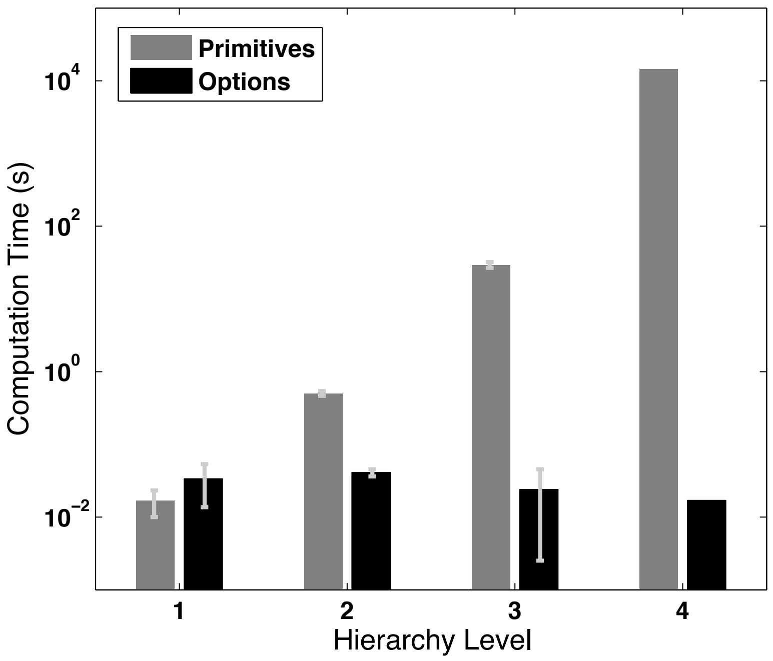

Figure 4.5. Policy computation times for tasks at varying levels of the Light Box hierarchy for an agent with primitive actions only and for one with options + primitives. Note the log scale.

4.6 Related Work

We outline here the related work on structure learning in FMDPs and intrinsically motivated RL that has the most in common with our approach. Although the literature on Bayesian network structure learning is substantial, many of these methods are not incremental and generally require that the data are drawn in an independent and identically distributed (i.i.d.) fashion (Abbeel et al., 2006). For the case in which we are interested, namely learning the structure of a DBN online from experience with an FMDP, these methods are thus not applicable. There are, however, a few incremental methods developed recently which search for DBN structures that best fit an agent's experience with an FMDP, where the data are not drawn i.i.d. because of the temporal dependencies involved. We review these approaches here and explain their advantages and disadvantages, justifying our choice to use the approach given in Jonsson and Barto (2007).

Strehl et al. (2007) present a unique incremental structure-learning algorithm that is actually composed of multiple instances of a "knows what it knows" (KWIK) algorithm (Li et al., 2008). The KWIK framework for self-aware learning is a formalism similar to the PAC formalism (Valiant, 1984) for analyzing the class of hypotheses learnable by a given supervised learning algorithm. A KIWK algorithm attempts to choose a hypothesis $h^* \in H$ that best fits a set of training examples provided to it by an environment, where $H \subseteq (X \rightarrow Y)$ is a set of probabilistic concepts, or hypothesis class, each element of which is a mapping from an input space X to an output space Y. The algorithm takes an input $x \in X$ and makes a prediction $\hat{y} \in Y \cup \{\perp\}$, where $\perp$ denotes an "I don't know" response, indicating that the algorithm is unable to make a good prediction of the true output y.

A hypothesis class H is KWIK-learnable by algorithm A if during a run of A's execution for any $\epsilon > 0$ and $\delta < 1$, with probability $1 - \delta$, A predicts for each input x either an output $\hat{y} \in Y$ that is within ϵ of the true output y, or predicts $\perp$ a total number of times that is bounded by a function polynomial in $1/\epsilon$ and $1/\delta$. The main point of these requirements is that if a hypothesis is KIWK-learnable by an algorithm A, then A is likely to find a hypothesis that makes near-optimal predictions given a polynomial number of data samples. The structure-learning algorithm in (Strehl et al., 2007) makes use of a set of KWIK algorithms to predict the value of each variable in the DBN representation of an FMDP's transition model given the previous state and action as input.

Although Strehl et al. prove that their algorithm has polynomial sample and computational complexity, their approach requires that the maximum number of parents D that a variable in the DBN may have be given a priori. In fact, the sample and computational complexity is exponential in D. This is because the algorithms keep statistics about each possible combination of values for each possible set of parents of size D or less. The structure-learning algorithm in this work was embedded in the Factored R_{max} framework (Guestrin et al., 2002), a factored version of the R_{max} algorithm (Brafman & Tennenholtz, 2003), which is a PAC-MDP approach for efficient exploration in MDPs. The authors' focus was therefore on efficiently achieving near-optimal behavior on a single task by balancing exploration with exploitation, not on learning modular solutions to ensembles of related tasks, as ours is.

Diuk et al. (Diuk et al., 2009) describe a novel KWIK structure-learning algorithm, called the adaptive k-meteorologists algorithm, that is more efficient than the algorithm presented in (Strehl et al., 2007), but whose computational and sample complexity is still exponential in D, the maximum in-degree of the DBN. This exponential dependence on D is unavoidable in provably optimal (or PAC) Bayesian network structure-learning (Abbeel et al., 2006). When D is large or no a priori information about the domain is known that allows one to specify a small D, these methods are therefore not feasible.

In contrast to these provably optimal approaches, there are greedy methods with only polynomial computational complexity that attempt to add dependencies to a DBN in an incremental fashion, and that have been shown to perform well empirically. They are, however, not guaranteed to find the best network. One such method is that of Jonsson and Barto (2007), which we use as a component of our framework and described earlier. Another is given by Degris et al. (2006), who present a structured form of the Dyna architecture for planning in MDPs (Sutton, 1991) which makes use of an incremental version of the SVI algorithm to handle planning and employs Utgoff et al.'s (1997) incremental tree induction (ITI) algorithm to learn the CPTs of an FMDP's transition and reward models online. A χ^2 test of significance between candidate conditional distributions is applied at the leaves of each of the trees to determine whether to split that leaf on a given variable at each time step. The approach is used to speed up learning on a single task by making use of offline computation to simulate actual experience. They, however, do not address skill learning or performance on ensembles of tasks.

Hart et al. (Hart et al., 2008) present an intrinsic reward mechanism that drives a bimanual robot to learn closed-loop, hierarchical control policies for various abstract behaviors (e.g., tracking, reaching, grasping). Their framework does not make use of the options formalism, but rather a similar scheme for closed-loop control in continuous dynamical systems, called the control basis. The control basis uses the convergence states of hand-engineered continuous controllers to produce a small, discrete state space in which standard RL algorithms may be applied. Intrinsic reward is given to the robot if the state of convergence of some controller that references an external set of stimuli switches from un-converged to converged, with the magnitude of the reward proportional to the number of externally referenced stimuli. This encourages the robot to learn behaviors that allow it to exercise specific types of stable control over its environment in various contexts. Although the learning of new skills is bootstrapped on existing skills, the addition of new skills in this work is controlled by the experimenter and not fully autonomous.

Mugan and Kuipers (2009) present a framework for autonomous learning of abstract skill hierarchies in continuous domains. The mechanisms for skill learning and abstraction they employ, however, apply only to discrete environments. They first discretize a continuous domain by extracting "landmarks," and then learn options to set the continuous variables that define the environment to values corresponding to these landmarks. The action (motor) variables are similarly discretized. They employ a modified version of DBNs as their representation of dynamics in terms of what they call "qualitative" variables and actions (the latter being options), but they do not utilize any of the structure-learning methods mentioned previously or any of the RL algorithms that exploit structural independence. The latter means that the option policies and value functions they learn using RL must be represented using a full lookup table, which in some cases can be much larger than the structured representations we employ. Additionally, while their focus is on learning skills applicable over ensembles of related tasks (and indeed there is no extrinsic reward in their framework), there is also no intrinsic reward. Rather, the agent simply chooses random actions (options) from its current skill set, which increases in complexity each time a new option is learned. Although they show that this can lead to increasingly complex behavior, there is no sense of the agent optimizing the rate at which this complexity increases.

Menashe and Stone (2015) propose a modification of our algorithm, replacing the options framework with what they call *transitions* (analogous to exits), and learning a transition graph which they use to do rough path planning through the state space. This graph takes the place of the models of temporally extended behaviors which the options framework provides. They then simulate multiple potential paths through the state space, sorted by their expected entropy gain (analogous to our intrinsic reward functions), and use the UCT algorithm (Kocsis & Szepesvári, 2006) to perform Monte-Carlo simulations of those paths using the agent's current transition model. This approach thus replaces the explicit planning we employ using the SVI algorithm with multiple sample trajectories generated from the agent's current model. The author's focus, however, is not on learning a hierarchical set of reusable skills for use on later exposure to novel tasks.

4.7 Summary and Future Work

We have presented an algorithm for autonomous, incremental learning of skill hierarchies in ensembles of finite FMDPs and active learning of domain structure using intrinsic rewards. Our results show that the construction of policies and models of abstract skills in this approach can provide drastic reductions in the computational costs of computing policies for novel but related tasks in a given domain when compared with costs using flat policy representations. The addition of options and associated planning methods into our scheme for active learning of environmental dynamics was shown to outperform previous methods of active structure learning that use only local information when guiding the agent to informative areas of its state space. This method of incremental option construction also makes our approach developmental in nature, allowing for steadily increasing behavioral complexity via bootstrapping of existing structural knowledge and behavioral expertise.

In both the approach presented in Jonsson and Barto (2006), and in our work, an agent constructs options to set every environmental variable to each of its possible values. For environments with large numbers of variables and/or values this may not be feasible or desirable. Rather one would like to consider ways of selectively constructing options based on some metric evaluating the utility of being able to set a variable to a certain value. In the case where the agent has a specific task this metric would likely take the task's reward function into account. In the initially task-less scenario we outline here, however, it is less clear what this metric should depend on. One possibility is to incorporate a designer-specified salience function that makes certain variable settings inherently more interesting to the agent than others (Barto et al., 2004).

Our choice of intrinsic reward was based largely on a previous method for active structure learning in FMDPs. This is clearly not the only possible intrinsic reward one could employ. Experimenting further with other ways to increase the rate at which new structure is acquired could yield new insights into more effective intrinsic rewards. Recent work has addressed searching in the space of reward functions for intrinsic rewards that result in faster learning (Lewis et al., 2010; Niekum et al., 2010). Perhaps methods such as these could be used to search for good intrinsic reward functions in our algorithm as well. Whatever form those rewards may take, however, they can be readily substituted into our developmental approach and make use of the incrementally increasing set of abstract skills generated by agents running our algorithm.

We took the approach of constructing potentially "premature" options because of the inability for our structure learning algorithm to distinguish between inherent domain stochasticity and incomplete structural knowledge. As a result we had to add parameters to our framework that will in general be domain-dependent. In the absence of domain knowledge this is undesirable, and so it would be fruitful to consider other methods for structure learning that could make this distinction reliably or to within some confidence factor without having to wait until the full structure of the domain is known. Although the alternative structure learning methods presented in Section 4.6 have this property, in the absence of prior domain knowledge their complexity is prohibitively high.

Finally, the mechanics of our current approach limits its applicability to finite FMDPs. Because certain components in our algorithm (e.g., the VISA algorithm) are not immediately extensible to the case of continuous states and/or actions, different techniques are required to produce autonomous, self-guided, developmental learning in structured environments with continuous state or action spaces. The following chapter takes a first step towards extending the principles of our developmental approach to such environments by presenting a novel algorithm for online, incremental structure learning of continuous FMDP models.

CHAPTER 5

INCREMENTAL STRUCTURE LEARNING IN CONTINUOUS FACTORED MDPS

In the previous chapter we introduced an algoirhtm for intrinsically motivated skill learning in FMDPs. The mechanism for structure learning we employed in our approach limited its applicability to finite FMDPs; i.e., FMDPs whose state variables have finite sets of values and which have a finite number of actions. Many interesting real-world problems, however, cannot readily be formulated as finite FMDPs because their state or action spaces are inherently continuous and not amenable to discretization. While it is beyond the scope of this thesis to present a full extension of our algoirithm for intrinsically motivated skill learning to continuous FMDPs, we present here preliminary work on an essential component of such a system; namely, a method for incrementally learning the structure and parameters of a continuous FMDP's transition model online from a single trajectory of experience.

5.1 Continuous Factored MDPs

Recall from Section 2.4 the details of the FMDP formalism when the action set is finite and the state variables have finite domains. We use the same notation as in the finite case, except where otherwise noted. Thus, each dimension of the state space is still represented as a random variable $S_i \in \mathbf{S}$. In contrast to a finite FMDP, however, the states and actions of a continuous FMDP with an n-dimensional state space and m-dimensional action space are vectors in $\Re^n$ and $\Re^m$, respectively. Although we previously represented the transition and reward models of finite MDPs with one

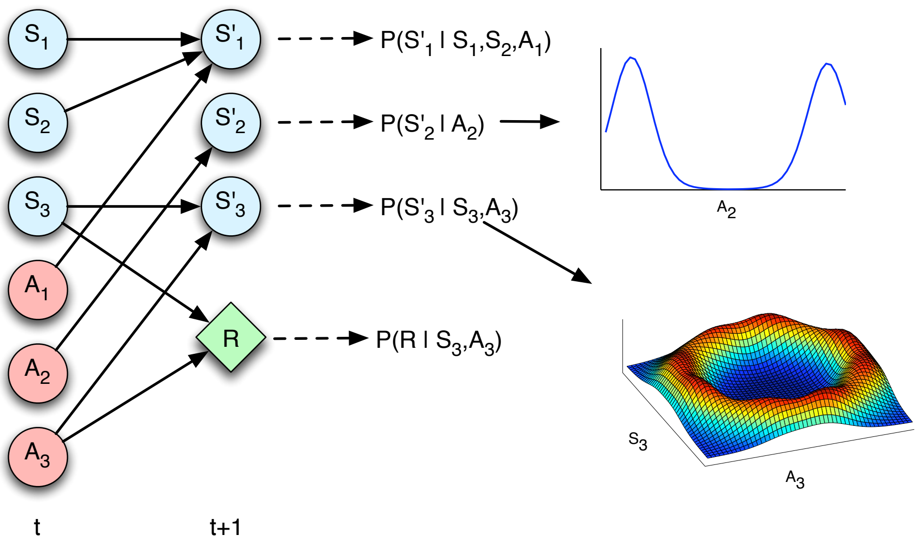

Figure 5.1. An example DBN for an FMDP with a 3-dimensional observation space and 3-dimensional action space. Example conditional probability distributions are shown for two variables.

DBN per action, this is not possible in continuous FMDPs. Rather, we represent these models using a single DBN with a set of action variables $\mathbf{A}$ that influence the state variables at the next time step, as shown in Figure 5.1.

As in the finite case, if we let $f_{\mathbf{X}}(\mathbf{s}, \mathbf{a})$ denote the projection of state-action pair $(\mathbf{s} \in \mathcal{S}, \mathbf{a} \in \mathcal{A})$ onto a set of variables $\mathbf{X}$ (i.e., the values the variables in $\mathbf{X}$ take on in $\mathbf{s}$ and $\mathbf{a}$), $f_{\mathbf{X}}(\mathbf{s})$ similarly denote the projection of state $\mathbf{s} \in \mathcal{S}$ onto $\mathbf{X}$, and $Par(Y)$ denote the set of parents of variable Y in the DBN, then the transition function P can be expressed in factored form as

$$P(\mathbf{s}'|\mathbf{s}, \mathbf{a}) = \prod_{i}^{n} P(S_i' = f_{S_i'}(\mathbf{s}')|Par(S_i') = f_{Par(S_i')}(\mathbf{s}, \mathbf{a})). \tag{5.1}$$

This form represents the transition function as the product of several conditional distributions, each of which may have much lower dimension than the full joint distribution being modeled. It is this fact that leads to the computational savings associated with these models. Note that the reward model R can be represented in a similar fashion (the diamond-shaped node in Figure 5.1). It remains to be shown how to represent the component conditional distributions that comprise the full model (right-hand side of Figure 5.1). In the finite case, we represented them as conditional probability trees, one for each variable, with internal nodes corresponding to the parents of the given variable and leaves containing probability mass functions corresponding to the conditional distributions. Since this is not possible in the continuous case, a variation of the formalism in terms of conditional density functions must be used. The following section describes the representation we adopt for the conditional density functions, and an incremental method for estimating these densities online. We then present a method for computing mutual information between sets of random variables modeled by these estimators, an essential component for our structure learning algorithm.

5.2 Online Incremental Density and Information Estimation

5.2.1 Density Estimation

There are many choices for the form of density function we may adopt to represent the factors of an FMDP's transition model. We choose the mixture of Gaussians (MOG) model for several reasons, the first being that there are existing methods for both online, incremental estimation of these models and for efficient computation of their entropies, which we make use of in the following section. Secondly, these models provide a natural, efficient way of obtaining conditional probabilities from the joint distributions they represent, which is useful when employing them in reinforcement learning algorithms. Additionally, it has been shown that given a sufficient number of components, the MOG model can represent arbitrary densities (Titterington et al., 1985).

A MOG model with k components gives the probability of a vector $\mathbf{x} \in \Re^n$ as

$$p(\mathbf{x}|\Theta) = \sum_{i=1}^{k} \pi_i p_i(\mathbf{x}|\theta_i), \tag{5.2}$$

where $\Theta = \{\theta_1, \dots, \theta_k\}$ is a set of parameter vectors, one for each component, π_i is the mixing coefficient (or prior) of component i, and $p_i(\mathbf{x}|\theta_i)$ is the component-conditional density of component i, which is given as

$$p_i(\mathbf{x}|\theta_i) = \frac{1}{(2\pi)^{n/2}|\Sigma_i|^{1/2}} exp\left[-\frac{1}{2}(\mathbf{x} - \mu_i)^T \Sigma_i^{-1}(\mathbf{x} - \mu_i)\right], \tag{5.3}$$

where $\theta_i = \{\mu_i, \Sigma_i\}$ represents the mean vector and covariance matrix of component i.

Learning the parameters of MOG models is a difficult problem with much work devoted to it. While many approaches use some form of the Expectation-Maximization (EM) algorithm for this task (Dempster et al., 1977), this solution does not lend itself to an efficient online setting, and has often been found to be oversensitive to parameter initialization. One alternative approach uses a modification to the Self-Organizing Map (SOM) neural network architecture, called the Self-Organizing Mixture Network (SOMN) (Yin & Allinson, 2001), to incrementally update the parameters of the model via a stochastic gradient-descent method. Although the SOMN is more general than a MOG model in that it can make use of non-Gaussian mixture components, we present here only the details of its operation for the case of Gaussian components.

After each new training example is observed, the parameters of a SOMN are updated so as to minimize the Kullback-Leibler divergence between the true (p) and estimated ($\hat{p}$) densities via stochastic approximation methods. Let $\hat{p}$ and $\hat{p}_i$ be the SOMN's estimates of (5.2) and (5.3), $\hat{\pi}_i^t$, $\hat{\mu}_i^t$, and $\hat{\Sigma}_i^t$ be the estimates of the prior, mean vector, and covariance matrix of component i, respectively, after t training examples have been observed, and $\hat{P}(i|\mathbf{x}) = \frac{\pi_i \hat{p}_i(\mathbf{x}|\theta_i)}{\hat{p}(\mathbf{x}|\Theta)}$ be the posterior probability of component i given training example $\mathbf{x}$. When a new training example $\mathbf{x}$ is observed, the parameters of each component are updated according to

$$\hat{\pi}_i^{t+1} = \hat{\pi}_i^t + \alpha[\hat{P}(i|\mathbf{x}) - \hat{\pi}_i^t] \tag{5.4}$$

$$\hat{\mu}_i^{t+1} = \hat{\mu}_i^t + \beta[\mathbf{x} - \hat{\mu}_i^t]\hat{P}(i|\mathbf{x}) \tag{5.5}$$

$$\hat{\Sigma}_i^{t+1} = \hat{\Sigma}_i^t + \gamma[(\mathbf{x} - \hat{\mu}_i^t)(\mathbf{x} - \hat{\mu}_i^t)^T - \hat{\Sigma}_i]\hat{P}(i|\mathbf{x}) \tag{5.6}$$

where $0 < \alpha, \beta, \gamma < 1$ are step size parameters. Alternatively, for computational efficiency a winning component may be selected based on the posterior probabilities, and only those components within a local neighborhood of the winner need be updated.

Experiments with the SOMN have shown that it is very robust to parameter initialization, and so it is common to initialize the priors evenly (i.e., $\pi_i = 1/k$, $\forall i$), the mean vectors randomly, and the covariance matrices to $\sigma\mathbf{I}$, where $\mathbf{I}$ is the identity matrix and σ is a scalar. The SOMN does require the number of mixture components k to be pre-specified, however, which is a significant limitation. We discuss possible remedies to this problem in our discussion section.

5.2.2 Mutual Information Estimation

Given the form of density function described above, we now present a method for incrementally estimating the mutual information between sets of random variables whose joint distribution is modeled by a SOMN. One way to express the mutual information $I(X, Y)$ between two (sets of) random variables X and Y is as a sum of entropies:

$$I(X, Y) = H(X) + H(Y) - H(X, Y), \tag{5.7}$$

where for a real-valued random variable X, $H(X) = -\int p(X) \log p(X) dX$ is the Shannon differential entropy of X. The joint differential entropy of random variables X and Y is given similarly as $H(X, Y) = -\int p(X, Y) \log p(X, Y) dX\ dY$.

Unfortunately, for our choice of density function, computing mutual information based on Shannon entropies is infeasible. However, for the case of a MOG density model as we have assumed, there is an approximation to Shannon entropy that has a particularly nice closed form. This is the quadratic Renyi entropy, a specific instance of a class of generalized entropies described by Renyi (1961), and for random variable X is given as $H_{R_2}(X) = -\int P(X)^2 dX$.

If we let $G(\mathbf{x} - \mu_i, \Sigma_i)$ represent the value of Gaussian mixture component i evaluated at $\mathbf{x}$, then note that $\int_X G(X - \mu_i, \Sigma_i) G(X - \mu_j, \Sigma_j) dX = G(\mu_i - \mu_j, \Sigma_i + \Sigma_j)$. Thus, for a mixture of k Gaussian densities, the quadratic Renyi entropy of the mixture density can be computed as

$$\begin{aligned} H_{R_2}(X) &= -\log \int P(X)^2 dX \\ &= -\log \int \left(\sum_i^k \pi_i G(X - \mu_i, \Sigma_i) \right)^2 dX \\ &= -\log \sum_{i=1}^k \pi_i \sum_{j=1}^k \pi_j G(\mu_i - \mu_j, \Sigma_i + \Sigma_j), \end{aligned} \tag{5.8}$$

so that the computation reduces to pairwise interactions between mixture components. Additionally, only half of these need to be computed in practice because of symmetry.

Although there is a similar closed form for the joint quadratic Renyi entropy of two random variables we could use to compute mutual information, we will take a slightly different tactic to obtain the joint entropy. Suppose we have a vector-valued random variable $X \in \Re^{r+s}$ so that $X = [X_1 \; X_2]^T$, $X_1 \in \Re^r$, and $X_2 \in \Re^s$. If $X \sim \mathcal{N}(\mu, \Sigma)$ is multivariate Gaussian with $\mu = [\mu_1 \; \mu_2]^T$ and $\Sigma = \begin{bmatrix} \Sigma_{11} & \Sigma_{12} \\ \Sigma_{21} & \Sigma_{22} \end{bmatrix}$ so that $\mu_1 \in \Re^r$, $\mu_2 \in \Re^s$, $\Sigma_{11} \in \Re^{r \times r}$, $\Sigma_{12} \in \Re^{r \times s}$, $\Sigma_{21} \in \Re^{s \times r}$, and $\Sigma_{22} \in \Re^{s \times s}$, then the marginal distribution of X_1 is itself a multivariate Gaussian distribution with mean

μ_1 and covariance Σ_{11}. The marginal of X_2 is similarly modeled with mean μ_2 and covariance Σ_{22}.

This means that we may obtain the joint entropy between two (sets of) variables X and Y by modeling their joint distribution explicitly, and their marginal entropies by applying (5.8) to the appropriate marginal distributions obtained from the joint, as just described. We will use this fact when evaluating potential dependencies in a DBN, as described in the following section.

5.3 Online Incremental Structure Learning

We now turn to our application of the techniques outlined so far to the problem of incremental structure learning in FMDPs with continuous states and actions. Previous work on learning factored transition models of finite FMDPs has taken the approach of adding dependencies to a DBN model of an FMDP one at a time when the mutual information between two variables is significantly high (Jonsson & Barto, 2007; Wolfe & Barto, 2006). We take a similar approach, although our techniques will be different because we are dealing with continuous states and actions.

Recall that mutual information can be expressed as a difference between the sum of two marginal entropies and their joint entropy, as in (5.7). Let $S_i'^{+} = S_i' \cup Par(S_i')$ be the union of a state variable $S_i' \in \mathbf{S}'$ in the DBN at time $t+1$ and its set of parents (in $\mathbf{S} \cup \mathbf{A}$) at time t. Our strategy will be to maintain, for each S_i', an estimate of the information between $S_i'^{+}$ and each other state and action variable in $\mathbf{S} \cup \mathbf{A}$ not already in $Par(S_i')$. We term each of these extra variables a *candidate* variable, and whenever the information between a candidate variable and $S_i'^{+}$ exceeds a pre-specified value, we add that variable to $Par(S_i')$ and remove it from the list of candidate variables for S_i'.

In order to do this we will maintain an estimate of the joint distribution of each possible combination of $S_i'^{+}$ and candidate variable $X \in (\mathbf{S} \cup \mathbf{A}) - Par(S_i')$. This initially requires the instantiation of n^2m SOMN models (nm models per state variable), where n and m are the dimensionalities of the state and action spaces of the FMDP, respectively. At time step t, each SOMN modeling a distribution containing S'_i is given a training example that is the concatenation of the values in the previous state and action vector corresponding to the current parents of S'_i and that distribution's associated candidate variable, and the value of S'_i in the current state vector. Initially this will result in the estimation of the joint distributions corresponding to each element of $\mathbf{S}' \times (\mathbf{S} \cup \mathbf{A})$. The techniques described in Section 3 now provide us with the means to compute the mutual information between each S'^{+}_i and each of its candidate variables from these joint distributions.

At every time step, after updating the density models, we evaluate each candidate variable X for each S'_i by computing the three entropies $H(S'^{+}_i)$, $H(X)$, and $H(S'^{+}_i, X)$ using (5.8) and the appropriate marginals obtained from the SOMN model for each $X \notin S'^{+}_i$, and then calculating the resulting information. For each S'_i, the candidate variable Y with the highest information gain above a pre-specified threshold η (if there is one) is added to the parents of S'_i (and thus to S'^{+}_i), and the SOMN modeling the joint distribution of S'^{+}_i and Y is removed from the set of SOMN models. Then, each of the other candidate distributions for S'_i is extended to incorporate Y by extending the mean vectors and covariance matrices of each component in each model by one dimension. The values of the parameters for the new dimension can be initialized in various ways. We describe the method we used in our experiments in Section 5. Algorithm 8 provides the details of our approach.

The reader may wonder why one would not just maintain n SOMN models, each modeling the joint distribution between a given S'_i and all of its possible parent variables, and then simply evaluate the information between its current set of parents and each other variable by using the appropriate marginal distributions. The reason we do not do this is that as the dimensionality of the SOMN models increases, the

Algorithm 8 LearnStructure

```
Initialization:
$\mathcal{M} \leftarrow \{\}$
for each $S'_i \in \mathbf{S}'$ do
    for each $X \in \mathbf{S} \cup \mathbf{A}$ do
        initialize a 2-dimensional SOMN $m_{S'_i,X}$ to model $p(S'^{+}_i, X)$
        $\mathcal{M} \leftarrow \mathcal{M} \cup m_{S'_i,X}$
    end for
end for
$\mathbf{s} \leftarrow$ initial state
Maintenance:
for t=1 to $\infty$ do
    $\mathbf{a} \leftarrow$ choose action
    $\mathbf{s}' \leftarrow$ next state
    for each $m_{S'_i,X} \in \mathcal{M}$ do
        concatenate $f_{S'_i}(\mathbf{s}')$, $f_X(\mathbf{s}, \mathbf{a})$, and $f_{Par(S'_i)}(\mathbf{s}, \mathbf{a})$ into training example $\mathbf{x}$
        update $m_{S'_i,X}$ with $\mathbf{x}$ using (5.4), (5.5), and (5.6)
        compute $I(S'^{+}_i, X)$ using (5.7) and (5.8) (see text)
        if $I(S'^{+}_i, X) > \eta$ then
            $\mathcal{M} \leftarrow \mathcal{M} - m_{S'_i,X}$
            $S'^{+}_i \leftarrow S'^{+}_i \cup X$ (add $X$ as parent of $S'_i$)
            for each $Y \notin S'^{+}_i$ do
                extend $m_{S'_i,Y}$ to model new dimension $X$ (see text)
            end for
        end if
    end for
    $\mathbf{s} \leftarrow \mathbf{s}'$
end for
```

accuracy of the density estimate becomes more difficult to maintain with relatively few mixture components. This is a consequence of the curse of dimensionality. The idea behind maintaining a larger number of low-dimensional models is to keep the number of mixture components necessary for an accurate estimate at a reasonable number. This is justified to some degree by the assumption that the environment we are trying to model does in fact contain structure in its dynamics, and that the number of parents of any given S'_i will in general be much smaller than the total number of state and action variables.

We would like to note, as we mentioned above, that our choice of the MOG model to represent the factors of the transition function P results in a very simple method for obtaining the conditional probability of a state $\mathbf{s}'$ given the previous state $\mathbf{s}$ and action $\mathbf{a}$. Note that for a multivariate Gaussian random variable $X = [X_1\ X_2]^T$ as defined in the example in Section 3, the conditional probability of X_1 given X_2 is also a multivariate Gaussian with mean $\mu_{1|2} = \mu_1 + \Sigma_{12}\Sigma_{22}^{-1}(X_2 - \mu_2)$ and covariance $\Sigma_{1|2} = \Sigma_{11} - \Sigma_{12}\Sigma_{22}^{-1}\Sigma_{21}$, where μ and Σ and their components are also defined as they were in the example in Section 3. Thus, since each S'_i is modeled jointly with its parents in the DBN by a MOG, one can obtain each component of $\mathbf{s}'$ given $\mathbf{s}$ and $\mathbf{a}$ by conditioning on its parents in this manner.

5.4 Experiments

We evaluated our approach on a structured environment which was actually multiple, independent instantiations of a single MDP whose state and action spaces were conglomerated into a single FMDP. The replicated MDP was a continuous "grid-world" with two state dimensions (horizontal and vertical position) and two action dimensions (horizontal and vertical movement). The state dimensions ranged from 0 to 1 and the action dimensions from -0.1 to 0.1, Each action dimension changed the position of the agent in the appropriate dimension by its amount plus some mean-zero

Gaussian noise with 0.01 standard deviation. All action dimensions were executed concurrently and so each action was vector-valued.

The state (action) vector for the full FMDP was constructed by concatenating the state (action) vectors of each MDP into a single vector. We then also added three dimensions to the resulting state vector that were independent of the dynamics of any of the component MDPs, and three action dimensions that had no effect on any of the state dimensions of the full FMDP. The three added state dimensions output at each time step, respectively, a random value in $[0, 1]$, a constant value (0.5), and a value (initialized to 0) that added Gaussian noise to it's previous value (with a mean of 0.05 and a variance of 0.001) and that wrapped around to 0 when the value reached 1. These extra state dimensions were just a few arbitrary ways of adding independent dimensions to the FMDP, though the final extra dimensions clearly depends on itself. The values of all state and action dimensions for each of the MDPs were normalized to be in $[0, 1]$ when provided as training samples to the individual SOMN models.

We initialized the SOMN models with 9 mixture components arranged in a regular grid over $[0, 1]^2$ and set the initial covariance matrices to $0.3\mathbf{I}$. When distributions were extended by a dimension, we set the values of the mean-vectors for the new dimension to be evenly spaced over $[0, 1]$ and set the last row and column of the covariance matrix to be all zeros, but with 0.3 in the last position. We set the threshold η to 3.0 in both experiments.

Figure 5.2 shows the number of correct dependencies as a function of time step for the first environment, averaged over 30 runs. Error bars show standard deviation. There are 13 dependencies in the correct model for this environment. Each of the 6 observation dimensions depends on itself at the previous time step and one action dimension, and the last extra dimension depends on itself. The curve shows that our algorithm was able to discover the correct structure in a reasonably short period of time. No incorrect dependencies were added by our algorithm at any point.

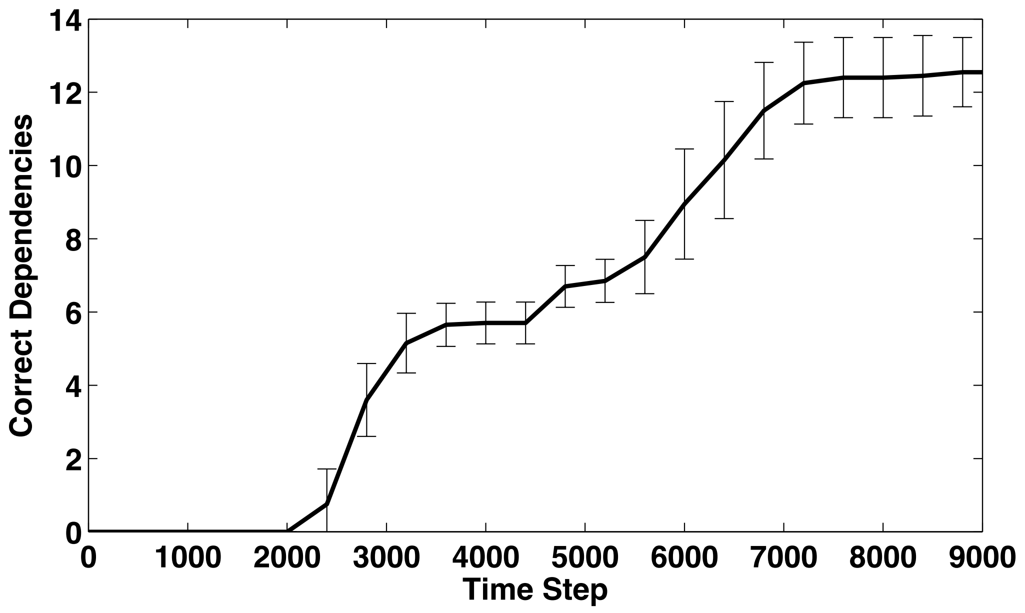

Figure 5.2. Results for the 3-MDP domain.

5.5 Summary and Future Work

We have presented a method for online, incremental learning of transition models of FMDPs with continuous state and action spaces. We use incremental density estimation techniques to model the factored transition function and information theoretic principles to add dependencies to a DBN model of the FMDP. Our experimental results show that our approach is able to discover the correct structure of a non-trivial, continuous, structured environment efficiently.

One limitation of our method is the use of the SOMN model, which requires a prespecified number of mixture components to model a given density. A possible remedy for this is the incorporation of methods used in other SOM approaches which allow the number of units in the map to vary as new data are received (Xiong et al., 2004). If successful, each mixture model need only use as many components as necessary to obtain an accurate density estimate, potentially reducing computation time.

The complexity of our approach is quadratic in the number of state and action dimensions, which poses a problem in very high-dimensional environments. It is possible, however, to incorporate prior knowledge about an environment either in the form of pre-specified dependencies or, less restrictively, constraints on the set of candidate variables considered. This could potentially reduce the computational load significantly. Additionally, the calculation of mutual information need not be done at every time step—only when one desires to evaluate candidate variables.

Our approach may be used in a feature selection setting for value function approximation in reinforcement learning, particularly in the case of learning abstract, temporally extended actions, or options (Sutton et al., 1999). Although traditionally much of the work on options has assumed the same state representation for each option in a given MDP, recent work has focused on the scenario in which each option has its own state abstraction (Jonsson & Barto, 2006; Konidaris & Barto, 2009). The possibility of state abstraction not only reduces the difficulty of learning a given option by reducing the number of variables over which the value function must be supported, but also increases the efficiency with which an agent learns to use an option by generalizing its value across states that differ along irrelevant dimensions.

The structural dependencies learned by our algorithm provide the subset of observation and action dimensions relevant to manipulating a particular set of dimensions in the environment. If the objective of an option is to set such a set of dimensions to a value in a specific range, then our approach provides an appropriate (reduced) subspace of the state-action space over which a value function may be approximated. That subspace will likely be significantly smaller than the full state-action space, greatly reducing the difficulty of learning that option.

CHAPTER 6

INCREMENTAL MODEL LEARNING IN CONTINUOUS DYNAMICAL SYSTEMS

All of the work presented thus far has dealt with model learning and intrinsically motivated skill acquisition in MDPs. There are, of course, many interesting problems that cannot be formulated as MDPs because they don't satisfy the Markov property—the observations the agent receives at a given time step are insufficient to perfectly disambiguate the state of the environment. As a first step toward extending our developmental framework to more general environments, this chapter presents an incremental algorithm for learning the parameters of a continuous TD network, which is a formalism for representing the state of a discrete-time, dynamical system with continuous actions and observations. The algorithm presented here is the first incremental algorithm for learning the parameters of a predictive state representation of a continuous dynamical system (Vigorito, 2009). We discuss the relevant details of dynamical systems and predictive state representations, specifically TD networks, in the following sections.

6.1 Dynamical Systems

We detail here two formalizations of controlled, discrete-event dynamical systems (DEDS), one with discrete observations and actions, and one with continuous observations and actions. We do not consider continuous-time systems in this work. A DEDS with discrete observations and actions consists of two finite sets of symbols, $\mathcal{O}$ and $\mathcal{A}$, the observations and actions, respectively, and a dynamics function P, described below. At each discrete time step t the system outputs an observation $o_t \in \mathcal{O}$ and accepts an action $a_t \in \mathcal{A}$. The alternating sequence of observations and actions that begins with the observation output at time 0 and ends with the action accepted at time t is the history $h_t \in \mathcal{H}$ of the system at time t, where $\mathcal{H}$ is an infinite set of all possible histories. Each observation is chosen probabilistically according to $P : \mathcal{O} \times \mathcal{H} \rightarrow [0, 1]$, where $P(o, h)$ gives the probability that $o_{t+1} = o$ for some $o \in \mathcal{O}, h \in \mathcal{H}$. Thus, P induces a probability mass function over $\mathcal{O}$, from which an observation is sampled at each time step.

A DEDS with continuous observations and actions is composed of two infinite sets, $\mathcal{O} \subseteq \Re^o$ and $\mathcal{A} \subseteq \Re^a$, again representing the observations and actions, where o and a are the dimensionalities of the observation and action spaces, respectively. As above, histories are alternating sequences of observations and actions, though in the continuous case each observation and action is a point in its associated vector space rather than a discrete symbol. The dynamics function P is defined in the same way, but now induces a probability density function over $\mathcal{O}$ from which each observation is sampled at every time step.

It should be noted that although the formalization of a discrete DEDS given above assumes a single observation is output by the system at each time step, it is trivial to extend this to vector-valued, discrete observations. Additionally, the formalization of an uncontrolled DEDS with either continuous or discrete osbervations is the same as above, but excludes the action set $\mathcal{A}$. Histories are then comprised of sequences of observations, and the definition of the dynamics function P remains the same in each case. In the following section we outline the background of predictive state representations and describe previous work with TD networks for modeling a discrete DEDS, after which we present our method for modeling a continuous DEDS using a continuous TD network.

First, it is worth noting that MDPs are a strict subset of DEDSs in which the most recent observation is sufficient to predict a sequence of future observations given a sequence of actions; i.e., they are DEDSs that satisfy the Markov property. For the sake of brevity when we refer to a DEDS in this chapter we specifically refer to those for which this is not the case, i.e., those which are partially observable, as these are the systems the work in this chapter is designed to address. Additionally, although the partially observable MDP (POMDP) is a formalism for MDPs that addresses the potential for partial observability in dynamical systems, it assumes that there is an underlying MDP generating the observations of the system, which we do not assume in our formalism.

6.2 Predictive State Representations

Predictive representations of state are a class of generative models that represent a dynamical system in terms of a set of predictions about sequences of observations generated by that system (Littman et al., 2002). Recent work has shown that certain formalizations of predictive representations are strictly more expressive than other models of discrete dynamical systems that use historical information or probabilistic distributions over unobservable variables as a representation (e.g., k-Markov models, POMDPs) (Singh & James, 2004). Empirically it has also been shown that in certain domains predictive representations can lead to better generalization than other representations (Rafols et al., 2005). In addition to this theoretical and practical appeal, predictive representations have the desirable property of being *grounded* in the sense that the representation is defined exclusively in terms of observable quantities. Though they share this property with other representations, such as k-Markov models, they are more expressive than such models.

One formalism for predictive representations is the Temporal-difference (TD) network (Sutton & Tanner, 2005). TD networks use well-established TD learning methods to incrementally update the predictions that define their state based on a stream of successive observations. All previous research with TD networks has focused on modeling dynamical systems with discrete observations and actions. Although there has been some work on other formalisms of predictive representations in continuous systems (Wingate, 2008), these approaches have not yet been extended to a fully online, incremental setting.

Since the developmental scenarios of interest in this thesis require fully incremental algorithms, the work presented here is intended to fill this gap, providing a method which can potentially be used to apply these principles to environments that can be represented as continuous, partially observable dynamical systems. We present an adaptation of the TD network formalization for making predictions in discrete dynamical systems that instead makes predictions about the values of feature functions defined over the observation space of a continuous dynamical system, as well as a method for conditioning those predictions on actions that also take on continuous values. In the following section, we outline the TD network formalism and describe previous work with TD networks in discrete dynamical systems. Section 6.4 presents our modification to the TD network architecture that supports continuous variables. We present results in noisy, continuous dynamical systems in Section 6.5 and discuss our findings and future work in Section 6.6.

6.3 TD Networks

Being a predictive representation, a TD network maintains state by updating at each time step the probabilities of a set of predictions, or *questions* about the system the network models. The semantics of these predictions are realized by a *question network*, which defines the TD relationships between different predictions. The question network is a set of nodes, each representing a specific scalar prediction about some observation of the system some number of time steps in the future.

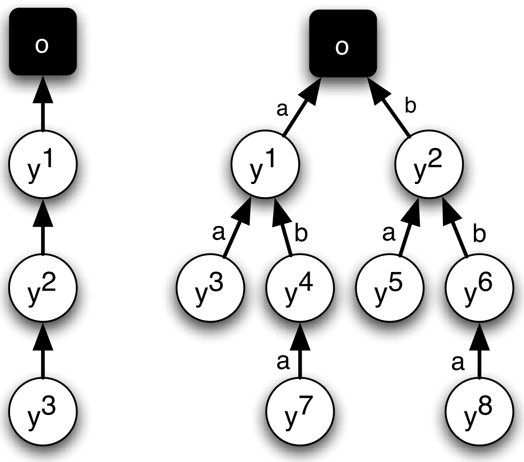

Figure 6.1. An example TD question network for an uncontrolled (left) and controlled (right) DEDS.

Example question networks for an uncontrolled and controlled dynamical system are shown in Figure 1.

The links between nodes in the question network provide the target for each prediction—the quantity it attempts to predict. This quantity may be defined in terms of another prediction (circles in Figure 6.3), observation data (squares), or both. For example, node y^1 in the left network of Figure 6.3 might make a prediction about the probability that the observation will be some specific value (e.g., 1 if the observation is binary) at the next time step.

If we let y^i_t denote the prediction of node y^i at time t, and z^i_t denote the target of y^i_t at time t, then in this example $z^1_t = Pr(o_{t+1} = 1)$. In contrast, node y^2 makes a prediction about the expected value of node y^1 at the next time step, and its target at time t is thus $z^2_t = E(y^1_{t+1})$. Note that this has the same meaning as predicting the probability that the observation will be 1 at time $t + 2$.

For controlled systems, predictions can be conditioned on actions, as seen in the right-hand network in Figure 6.3. The system modeled by this network has two actions, a and b, and action conditional predictions are indicated via labels on the links of the network. For example, node y^1 makes a prediction about the value of the observation at time $t+1$ given that the agent takes action a at time t. Similarly, node y^4 makes a prediction about the value of the observation at time $t+2$ given that the agent takes action b and then action a starting at time t. As in previous work, for ease of exposition we discuss only single-target question networks, i.e., nodes with only one parent.

The actual values of the predictions semantically defined by the question network are computed by a separate function approximator called the answer network. The state of the system, or output of the answer network is the vector of predictions $\mathbf{y}_t \in \Re^n$, where n is the number of nodes in the network. At each time step an input vector $\mathbf{x}_t$ is computed as some function of the previous predictions $\mathbf{y}_{t-1}$, the previous action a_{t-1}, and the newly received observation o_t:

$$\mathbf{x}_t = \mathbf{x}(\mathbf{y}_{t-1}, a_{t-1}, o_t) \in \Re^m. \tag{6.1}$$

The prediction vector $\mathbf{y}_t$ is then computed as some function $\mathbf{u}$ of $\mathbf{x}_t$ and a modifiable parameter $\mathbf{W}$:

$$\mathbf{y}_t = \mathbf{u}(\mathbf{x}_t, \mathbf{W}) \in \Re^n. \tag{6.2}$$

A stochastic gradient descent update rule is used to modify the weights w^{ij} of the network according to

$$\Delta w^{ij} = \alpha(z_t^i - y_t^i)c_t^i \frac{\partial y_t^i}{\partial w^{ij}}, \tag{6.3}$$

where α is a step size parameter, c_t^i is an action condition defined below, and z_t^i is the i^{th} element of the target vector $\mathbf{z}_t$, which is computed as some function $\mathbf{z}$ of the latest observation and predictions:

$$\mathbf{z}_t = \mathbf{z}(o_t, \mathbf{y}_t) \in \Re^n. \tag{6.4}$$

The action conditions $\mathbf{c}_t$ determine the degree to which each prediction is responsible for matching its target given the agent's behavior, as defined by the question network. Formally, $\mathbf{c}_t$ is defined to be some function $\mathbf{c}$ of the previous action and predictions

$$\mathbf{c}_t = \mathbf{c}(a_{t-1}, \mathbf{y}_{t-1}) \in [0,1]^n. \tag{6.5}$$

Tanner and Sutton (2005) introduced TD(λ) networks, which incorporate eligibility traces to deal with certain shortcomings of conventional TD networks. Eligibility traces were originally introduced in Sutton (1988) to provide a mechanism for making more general n-step backups of predictions in conventional TD learning, rather than the traditional 1-step backups. The parameter $\lambda \in [0,1]$ controls the degree to which longer sequences of predictions act as a target for learning. When $\lambda = 0$, the target is simply the 1-step prediction. When $\lambda = 1$, the longest possible sequence of predictions is used as a target and given the full weight of each update. Intermediate values of λ result in exponentially weighted averages of sequences of varying lengths being used as targets.

Notation for node targets in TD(λ) networks makes use of the parent function $p(i)$, which denotes the parent of node y^i as defined by a single link in the question network. Later parents of a node are denoted as $\{p(p(i)), p(p(p(i))), \ldots\}$, or $\{p^2(i), p^3(i), \ldots\}$ in short form, so that $p^k(i)$ identifies the k^{th} parent of node y^i. The machinery necessary for incorporating eligibility traces is slightly more complicated for TD(λ) networks than for the TD(λ) algorithm used for value-function learning. A trace for each prediction y_t^i must be maintained, and at each step the algorithm checks to see that the sequence of recently executed actions matches the conditions for the prediction, eliminating the trace if this is not the case.

Given a trace initialized at time step $t-k$ whose action conditions over the last k time steps have been satisfied, the weights $\mathbf{W}$ are updated at time step t using the temporal difference information $\mathbf{y}_t - \mathbf{y}_{t-1}$ and the past input vector $\mathbf{x}_{t-k}$ to improve the past prediction $\mathbf{y}_{t-k}$, with the update scaled appropriately by λ^{t-k-1}. We have only presented essential notation and intuition here, and refer the reader to Tanner and Sutton (2005) for the full details of the algorithm. In the following section we present our modified TD(λ) algorithm, which allows for continuous observations and actions.

6.4 Continuous TD Networks

In the case of a dynamical system with continuous observations, one can no longer construct question networks to specify predictions of all possible values of the system's observations, since this would require infinitely many predictions. The solution we employ is to make predictions about the expected values of a set of feature functions defined over the observation space. More formally, we maintain a set Φ of feature functions $\phi_i : \Re^o \rightarrow \Re$, each element of which outputs at time t a scalar value $\phi_i(o_t)$, where o_t is the observation at time t and o is the dimensionality of the observation space. Predictions of the values of these functions, which together define state, can then be used as features for approximating other functions, e.g, value functions in a reinforcement learning setting (Sutton & Barto, 1998).

Figure 6.4 shows an illustration of a possible question network for an uncontrolled DEDS with continuous observations. The feature functions in this case can be thought of as radial basis functions with spherical covariance matrices evenly tiled over a two-dimensional observation space. Each feature function acts as a target observation and each node predicts the expected value of one of the functions some number of time steps in the future. The semantics of the links are the same as those defined for discrete TD networks.

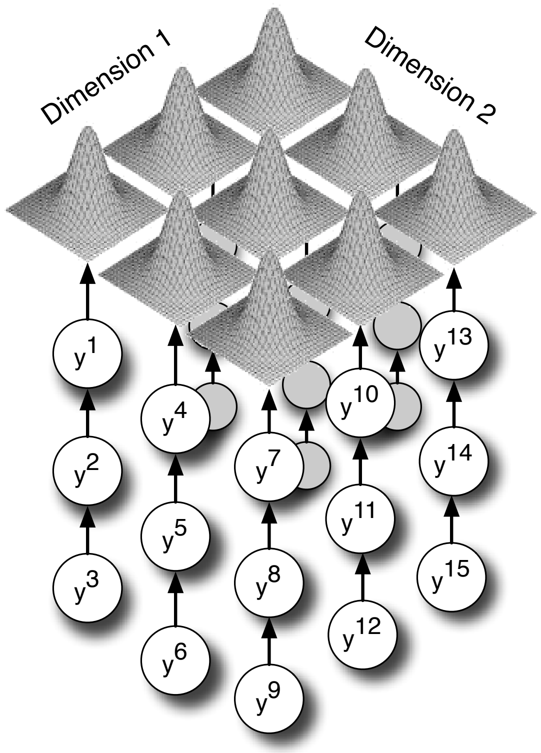

Figure 6.2. A example TD question network for an uncontrolled DEDS with continuous observations.

We must also provide a method for dealing with continuous actions in this framework, since it is not possible to have infinitely many action conditions in the question network. In all previous work with TD networks the action conditions were assumed to be binary since the action set was finite and small. The definition of the action conditions given in (6.5), however, is more general and allows for real-valued action conditions between 0 and 1.

To handle action conditions that allow for generalization over similar actions, we assume a set Ψ of activation functions $\psi_i : \Re^a \to \Re$, each element of which takes an action in $\Re^a$ and provides a scalar value in $[0, 1]$ according to some similarity metric, indicating the degree to which that action matches the associated activation function. Each link of the question network may thus be conditioned on a particular activation function $\psi_i \in \Psi$ just as links are conditioned on specific actions in a discrete TD network. The value of the action condition for node y^i at time t is computed as $c_t^i = \psi_i(a_{t-1})$, where ψ_i is the activation function on which node y_i is conditioned.

We use Euclidean distance as a similarity metric in this work and employ radial basis functions as our activation functions for their ease of use. The action space is tiled evenly by these functions, with each ψ_i having a different center $\mu_i \in \Re^a$ and $a \times a$ covariance matrix $\mathbf{\Sigma}$. The value of each ψ_i given action a_t is thus computed as

$$\psi_i(a_t) = e^{-(\mathbf{a}_t - \mu_i)^T \mathbf{\Sigma}_i^{-1} (\mathbf{a}_t - \mu_i)/2}. \tag{6.6}$$

Although this allows for general covariance matrices, in our experiments we use spherical covariance matrices so that $\mathbf{\Sigma} = \sigma \mathbf{I}$, where $\mathbf{I}$ is the identity matrix and σ is a parameter that determines the kernel width.

Algorithm 9 shows the TD(λ) network learning algorithm for systems with continuous observations and actions. The algorithm is modified from Tanner and Sutton (2005). The first major difference is the construction of the answer network's input vector $\mathbf{x}_t$. Our approach constructs $\mathbf{x}_t$ by concatenating $\mathbf{y}_{t-1}$ with a vector contain-

Algorithm 9 Psuedo-code for the TD(λ) network learning algorithm for continuous dynamical systems. Modified from Tanner and Sutton (2005).

```
$\Phi \leftarrow$ set of observation feature functions
$\Psi \leftarrow$ set of action activation functions
$\mathbf{y} \leftarrow$ initial state vector
$\mathbf{W} \leftarrow$ initial weight matrix
$Traces \leftarrow \{\}$
for $t$=0 to $T$ do
    $newTraces \leftarrow \{\}$
    $\mathbf{a} \leftarrow chooseAction()$
    $\mathbf{o} \leftarrow getObservation(\mathbf{a})$
    $\mathbf{x}_t \leftarrow x(\mathbf{a}, \mathbf{o}, \mathbf{y}_{t-1}, \Phi, \Psi)$
    $\mathbf{y}_t \leftarrow \mathbf{W}\mathbf{x}_t$
    for $(i, k) \in Traces$ do
        if $p^{t-k}(i) \neq observation$ then
            $z \leftarrow \mathbf{y}_t[p^{t-k}(i)]$
        else
            $z \leftarrow \phi_{p^{t-k}(i)}(o)$
        end if
        $p \leftarrow \mathbf{y}_{t-1}[p^{t-k-1}(i)]$
        $c_k \leftarrow traceCondition(i, k) \cdot \psi_{p^{t-k-1(i)}}(\mathbf{a})$
        for $w^j \in W[i]$ do
            $w^j += \alpha(z - p)c_k x_k^j \lambda^{t-k-1}$
        end for
        if $p^{t-k}(i) \neq observation$ then
            $traceCondition(i, k) \leftarrow c_k$
            $newTraces \leftarrow newTraces \cup (i, k)$
        end if
    end for
    for $i \in \mathbf{y}$ do
        $traceCondition(i, t) \leftarrow 1$
        $newTraces \leftarrow newTraces \cup (i, t)$
    end for
    $Traces \leftarrow newTraces$
end for
```

ing the values of the observation basis functions $\phi_i \in \Phi$ and of the action activation functions $\psi_i \in \Psi$ at time t given $\mathbf{o}_t$ and $\mathbf{a}_{t-1}$. The size of the input vector is thus $|\mathbf{y}| + |\Phi| + |\Psi|$. This is in contrast with previous work in which the input vector contained binary elements corresponding to each possible action-observation pair.

The answer network in previous work with TD networks was implemented as a generalized linear model, so that

$$\mathbf{y}_t = \sigma(\mathbf{W}\mathbf{x}_t) \in \Re^n, \tag{6.7}$$

where the parameter $\mathbf{W}$ was a $|\mathbf{y}| \times |\mathbf{x}|$ weight matrix, and σ was the vector-valued logistic function $\sigma(x) = \frac{1}{1+e^{-x}}$ applied element-wise to $\mathbf{W}\mathbf{x}_t$. In our work, however, nodes do not predict probabilities of binary predicates corresponding to discrete observations, and so the use of the logistic function to filter $\mathbf{W}\mathbf{x}_t$ is not appropriate. We thus let σ be the identity function, resulting in a simple linear function approximator.

The final distinction concerns the method of updating eligibility traces. Because we use non-binary action conditions, traces are not eliminated as they are in the discrete algorithm when action conditions fail. Rather the action condition values must appropriately weight the updates associated with each trace based on the agent's recent actions. In order to achieve this, each trace must store an accumulated action condition that is initialized to 1 when the trace is created, and updated at each time step by multiplying it by the action condition value at the current time step. The function *traceCondition*$(\cdot)$ in Algorithm 9 represents the action condition value currently associated with a given trace.

6.5 Experiments

To evaluate our approach we tested our algorithm on a small set of partially observable, continuous dynamical systems. The systems are partially observable in the sense that the most recent observation does not provide enough information to maintain state; i.e., it is not a sufficient statistic for history. Multi-step predictions are thus needed to model the systems accurately.

For all of our experiments we employed radial basis functions (RBFs) for our observation features Φ as well as for our set of action activation functions Ψ. The action conditions were thus computed as given in (6.6), and the observation features, similarly, as

$$\phi_i(o_t) = e^{-(\mathbf{o}_t-\mu_i)^T \mathbf{\Sigma}_i^{-1}(\mathbf{o}_t-\mu_i)/2}, \tag{6.8}$$

where o_t is the observation at time t. The feature and activation function centers were tiled evenly over the observation space and action space, respectively, so that a system with observation dimension o and action dimension a had n^o feature functions and m^a activation functions, where n and m are the number of functions used per dimension for the observation and action spaces, respectively. As mentioned above we used spherical covariances for each of the functions so that $\mathbf{\Sigma} = \sigma\mathbf{I}$. We report the value of n, m, and σ for each experiment as it is discussed.

Although one would ideally like to automate the construction of the question network when learning TD networks, to keep clear our focus on learning the parameters of a TD network for a continuous system we have left the issue of question network discovery in continuous TD networks to future work. The choice of question network for each system was thus made according to intuition and some trial and error based on knowledge of the systems being modeled.

Though we experimented with a few different types of question networks, we wound up using ones of the form shown in Figure 6.3 for each system. Each feature function $\phi_i \in \Phi$ was the parent (target) of $|\Psi|$ nodes, each of which, conditioned on a distinct $\psi_j \in \Psi$, predicted the value of ϕ_i one time step in the future. We did not implement a fully conditional network (which would require a number of nodes exponential in $|\Psi|$), but rather found that successive, identical action conditions

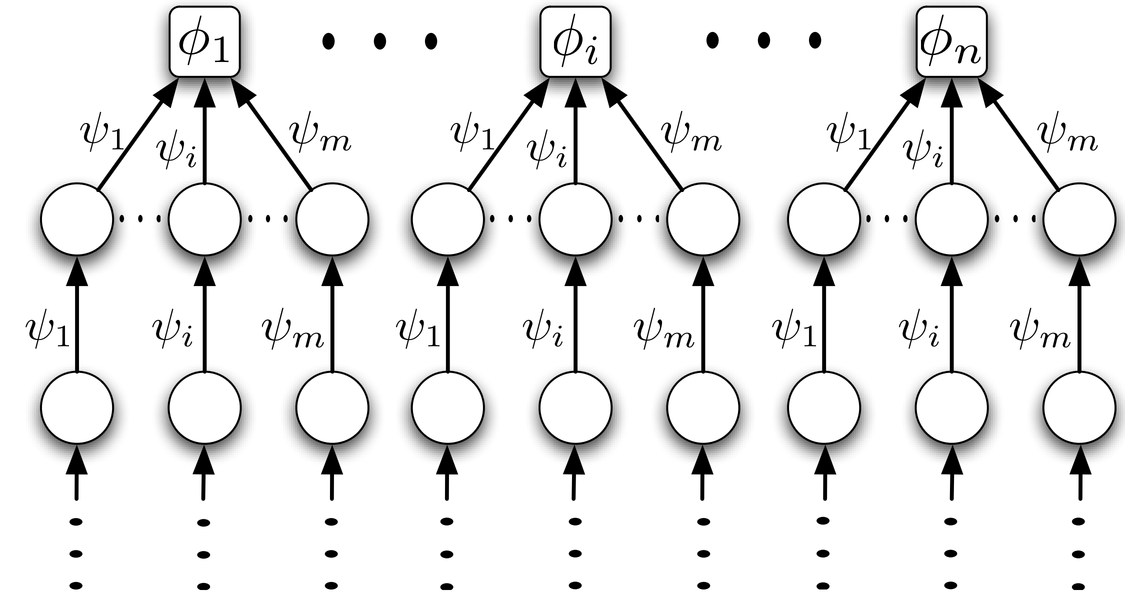

Figure 6.3. Form of the question networks used in our experiments. $|\Phi| = n$. $|\Psi| = m$.

chained off of each of the children of the observation nodes, as diagrammed in Figure 6.3, performed well. This structure is similar to previous question network structures used in some discrete TD networks (Tanner & Sutton, 2005). The trailing dots at the bottom of the figure indicate that the depth of each chain can vary. We used the same chain depth d for each chain in a given question network, and report that depth for each experiment below. All experiments used a step size parameter $\alpha = 0.01$ and eligibility parameter $\lambda = 1$.

6.5.1 Uncontrolled Systems

We first tested the ability of a TD network to learn models of uncontrolled dynamical systems. Since there are no actions in these systems the question networks look as they would in Figure 4 if $|\Psi| = 1$, so that each basis function ϕ_i has just one chain of d descendants. The first system was a simple square wave, which alternated between emitting one-dimensional values 0 and 1, each for five times steps at a time.

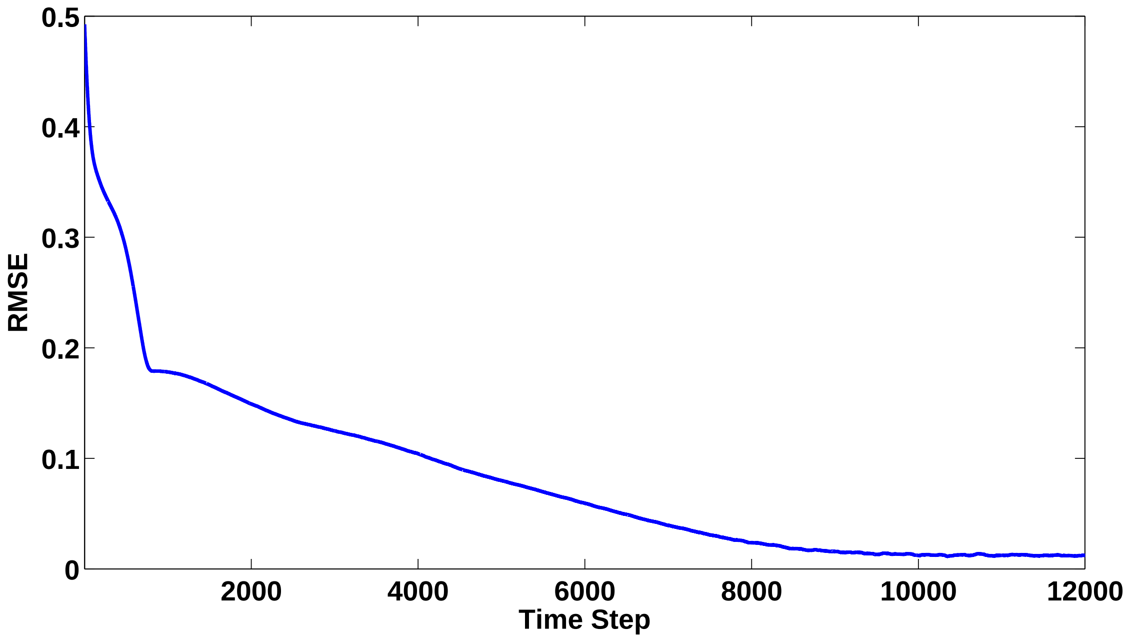

Figure 6.4. RMSE of the one-step predictions of all $\phi_i \in \Phi$ as a function of time for the noisy, uncontrolled square wave. Each point is an average of the error over the previous 100 time steps.

Each observation emitted by the system was corrupted by mean-zero Gaussian noise with standard deviation 0.05. This is essentially a noisy, continuous analog of the cycle world presented in Tanner and Sutton (2005). We let $n = 4$, and $\sigma = 0.3$ for this experiment. In order to maintain state the network must have at least five steps of prediction, and so we used a depth $d = 5$ for each chain.

Figure 6.4 shows the average root-mean-squared error (RMSE) of the one-step predictions of the expected values of each feature function at each time step, averaged over all $|\Phi|$ predictions. The errors were computed by taking the difference between the predicted values of the basis functions at a given time step and the actual observed values of those functions at the next time step, squaring those errors, averaging them over all feature functions, and taking the square root of that average. Each point in the graph is an average of the RMSE for the previous 100 time steps. The curve represents an average of 30 runs. We see that the network is able to learn a good

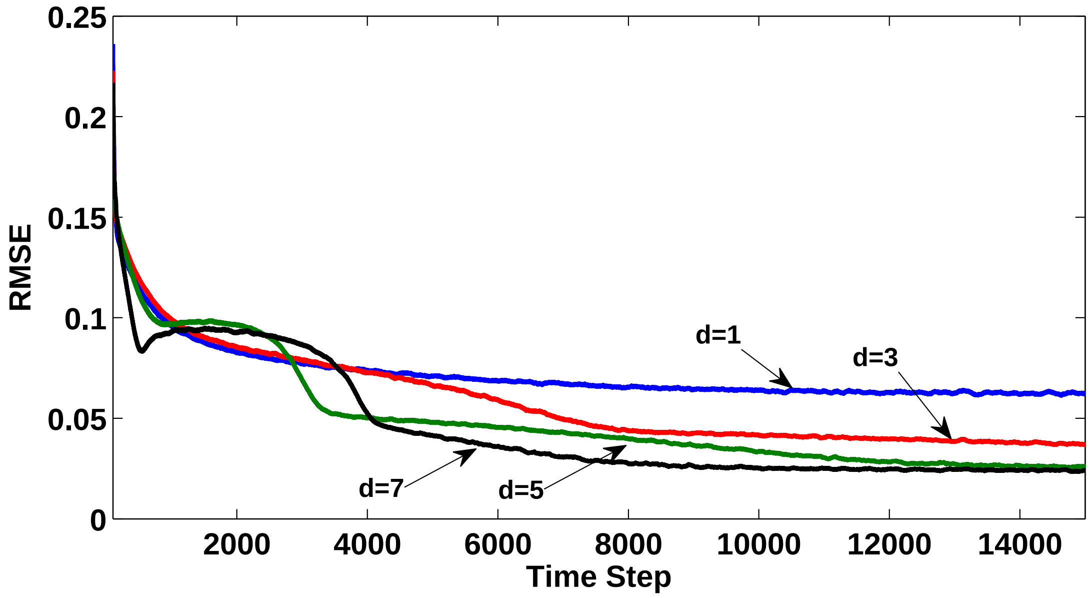

Figure 6.5. RMSE of the one-step predictions of all $\phi_i \in \Phi$ as a function of time for the noisy, uncontrolled sine wave. Each point is an average of the error over the previous 100 time steps.

model even given noisy observations with an amount of experience roughly equivalent to the amount taken to learn the deterministic, discrete version of this problem, as presented in Tanner and Sutton (2005).

We next experimented with predicting a sinusoid, where the observation emitted at time t was given by $o_t = (\sin(0.5t) + 1)/2$. Observations were again corrupted by mean zero Gaussian noise with 0.05 standard deviation. We again let $n = 4$, and $\sigma = 0.3$, but rather than pick a specific chain depth d of the question network for which to present results, we plot the learning curves for a few values of d. Each curve is again an average of 30 runs. Figure 6.5 shows these curves, and it is clear that while increasing the depth of the question network chains up from 1 through 5 improves the quality of the model learned, having depth greater than 5 does not produce very significant performance benefits aside from some slightly faster convergence.

6.5.2 Controlled Systems

We next evaluated our approach on two controlled versions of the dynamical systems used above. In each case, we introduced an action dimension that varied the amplitude of the corresponding wave function. The possible actions for each system were in $[0, 1]$, and resulted in modulating the amplitude from 0 to 1 continuously. That is, for a given action $a \in [0, 1]$, the square wave alternated between emitting values $a + \frac{1-a}{2}$ and $\frac{1-a}{2}$, each five steps at a time. Similarly, the sine wave emitted an observation at time t according to $o_t = \frac{a}{2}(\sin(0.5t) + 1) + \frac{1-a}{2}$, given action $a \in [0, 1]$. Observations in both systems were again corrupted by mean-zero Gaussian noise with standard deviation 0.05.

In both experiments we let $n = m = 4$, but we found it necessary to use different values of σ for the feature functions than for the action activation functions. We set the former, σ_ϕ, to 0.3, and the latter, σ_ψ, to 0.1. We again set the depth d of the question network chains to 5 in the square wave experiment, and varied the depth of the networks in the sine wave experiment, plotting the results for each depth.

The policies used to collect data were smoothed versions of a random walk over the action space. Errors were computed as above, where the expected values of a given feature function ϕ_i were calculated by weighting the predictions of each of the children of ϕ_i by the activation of the child's associated action activation function, given the last action taken.

Figures 6.6 and 6.7 show the RMSE of one-step predictions for the controlled square wave and sine wave experiments, respectively. We see in Figure 6.6 that the system is able to learn almost as good a model of the controlled system as it did of the uncontrolled system, indicating that our mechanism for handling continuous actions is viable. Similarly, Figure 6.7 shows that, although still dependent on having enough steps of prediction, the network is able to learn a good model of the controlled sine wave system as well.

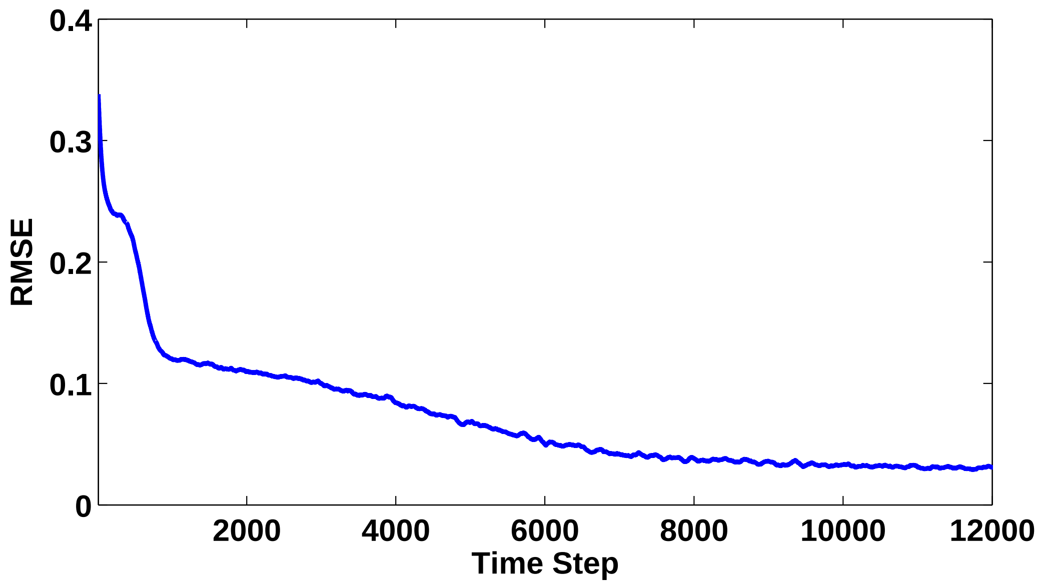

Figure 6.6. RMSE of the one-step predictions of all $\phi_i \in \Phi$ as a function of time for the noisy, controlled square wave. Each point is an average of the error over the previous 100 time steps.

Lastly, we tested our algorithm on a partially observable version of a dynamical system common in the reinforcement learning literature, the mountain car, in which an underpowered car must be driven up a steep cliff in a valley. Because the car is underpowered it cannot drive directly up the hill, but must reverse up the rear side of the valley to gain enough momentum to make it up the far side. We refer the reader to Sutton and Barto (1998) for the details of the dynamics. When the position and velocity of the car are given as observations, the system is fully observable. We thus eliminated the velocity component of the observation, producing a one-dimensional observation which is not sufficient to maintain state. Additionally, as in our other experiments, we corrupted each observation with mean-zero Gaussian noise with standard deviation 0.05.

As in the previous experiments, we let $n = m = 4$ and set σ_ϕ to 0.3 and σ_ψ to 0.1. The chain depth of the question network was set to 5. Figure 6.8 plots the RMSE

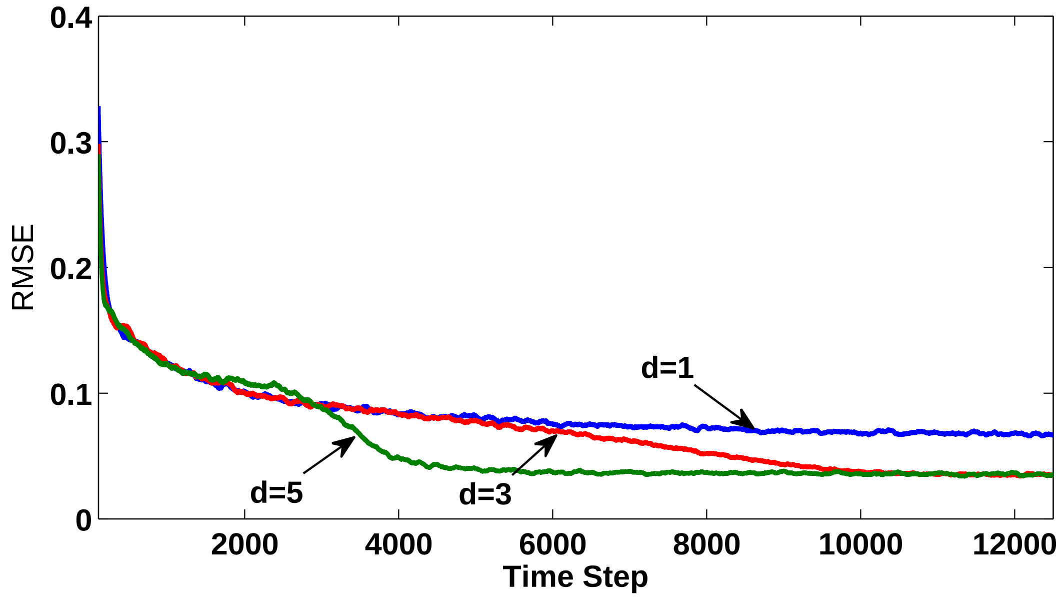

Figure 6.7. RMSE of the one-step predictions of all $\phi_i \in \Phi$ as a function of time for the noisy, controlled sine wave. Each point is an average of the error over the previous 100 time steps.

of one-step predictions for the mountain car system. We see that the TD network used was able to learn a good model of the dynamics and thus was able to recover the velocity dimension by making use of multi-step predictions.

6.6 Summary and Future Work

We have presented an extension to the TD(λ) network learning algorithm that is capable of modeling partially observable, noisy, continuous dynamical systems. Our algorithm represents the first instance of a fully incremental algorithm for learning a predictive representation of a continuous dynamical system (Vigorito, 2009). While there has been work on other formalisms of predictive representations in continuous systems (Wingate, 2008), these approaches are not incremental and thus would not be applicable to the developmental learning scenario that is the focus of this thesis.

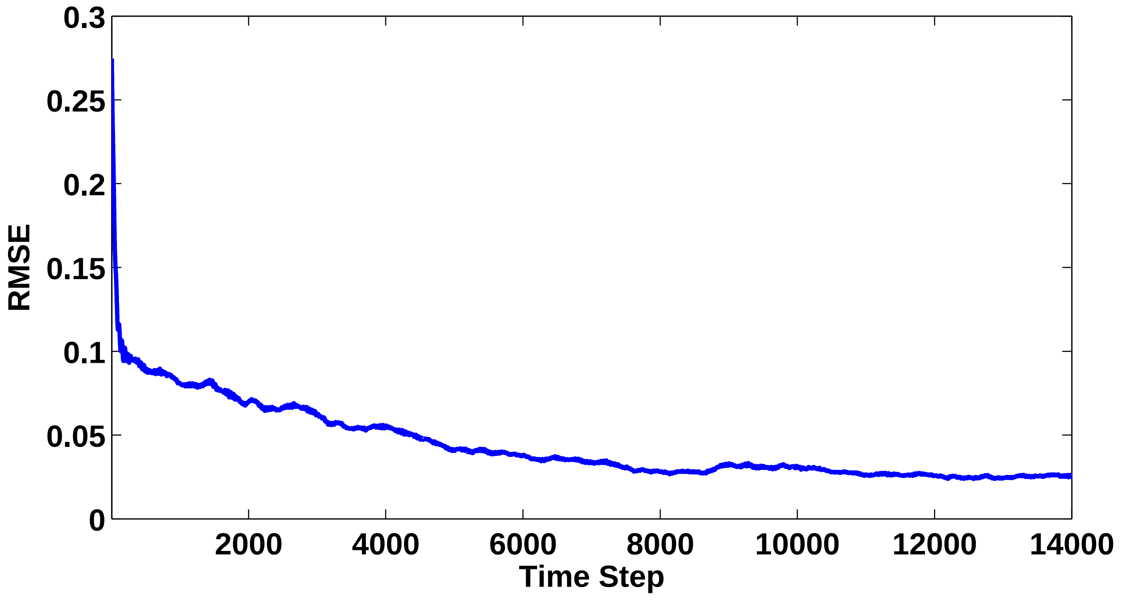

Figure 6.8. RMSE of the one-step predictions of all $\phi_i \in \Phi$ as a function of time for the noisy, partially observable mountain car task. Each point is an average of the error over the previous 100 time steps.

Our results show that our algorithm is capable of learning accurate models of both controlled and uncontrolled versions of such systems that are robust to noise.

In the work presented here we constructed question networks based on intuition and trial-and-error. In general it is desirable to automate this process and discover a (preferably minimal) question network based on a stream of experience. An online discovery algorithm for discrete TD networks was presented in Makino and Tagaki (2008). We chose not to employ it in this work so as to better isolate the contributions of extending TD networks to continuous systems. However, we see no immediate reason why the approach presented there cannot be combined with our extension.

One interesting direction for future research is the application of state abstraction to continuous TD networks to facilitate scaling our approach up to higher-dimensional systems. State abstraction in other predictive representation formalisms has been considered for both discrete (Wolfe et al., 2008) and continuous dynamical systems

(Wingate, 2008). However, although temporal abstraction in discrete TD networks has been explored recently (Sutton et al., 2006), to our knowledge there has been no work on state abstraction in TD networks. Using features that are defined over every dimension of the observation space is not always necessary for structured environments. Taking advantage of such structure may lead to compact representations that are easier to learn.

Finally, our approach has been agnostic about the observation features used. The accuracy of the model learned will obviously be dependent upon the form those features take. There is a large body of recent work on basis function selection and construction for value function approximation in Markov decision processes (Mahadevan, 2008; Parr et al., 2008), and it is interesting to consider applying work in those areas to choosing or constructing appropriate features for observation spaces in partially observable, continuous domains.

CHAPTER 7
CONCLUSIONS

The primary objective of this thesis was to present algorithms that achieve long-term, developmental learning of skill hierarchies in model-based reinforcement learning agents, and to show how this learning yields versatile, robust agents capable of solving many different problems in their environment more efficiently than solving them individually. We did this in part through the use of intrinsic rewards to guide the agent to the most informative parts of the state space given its current skill set and knowledge of the environment. A second objective was to present model-learning methods for applying these algorithms to environments that can be modeled with more sophisticated representations than traditional MDPs, specifically factored MDPs and partially observable dynamical systems.

To illustrate the performance of these methods, we described and experimented with the kinds of environments in which they exhibit the greatest advantages—those that are hierarchically organized and in which random exploration is unlikely to yield efficient exploration. Our methods allow agents in these environments to focus their exploration on the most informative areas of the state space, and as such to maximize their rate of skill acquisition. In fields like robotics where accurate simulators are not easily implemented and thus sampling arbitrary portions of the state space is difficult, these methods are particularly appropriate.

Finally, we demonstrated the utility of an agent running our algorithm for an initial developmental period when later faced with various extrinsically rewarded tasks by testing their ability to solve novel problems in their environments after this period.

The following section summarizes the specific contributions of this thesis, after which we discuss the limitations of our methods and potential future work.

7.1 Summary of Contributions

7.1.1 Intrinsically Motivated Skill Learning in Markov Decision Processes

We presented an algorithm for long-term, incremental learning of abstract skill hierarchies in environments formally represented as MDPs. We showed how an agent in this framework can be intrinsically motivated to learn increasingly complex behaviors by continually improving models of the effects of its actions on its environment and incrementally creating skills based on those models to increase its breadth of control. Intrinsic reward functions, which we defined in terms of an agent's internal state, focus the agent's exploration efforts on areas of the environment in which its model is inaccurate, but which are reachable with its current skill set. This form of active learning leads to a developmental process that bootstraps skill learning and model learning using the agent's current predictive and procedural knowledge. We showed that, in certain classes of environments, our methods allow for acquisition of complex behaviors not efficiently achievable by random exploration methods. Finally, we showed that the acquisition of these skill hierarchies renders agents in our framework able to more efficiently solve novel tasks posed to them than learning from scratch.

7.1.2 Intrinsically Motivated Skill Learning in Factored MDPs

The traditional MDP formalism suffers from the curse of dimensionality, with the state space increasing exponentially in the number of variables that define it, even if some of these variables are independent of each other. Taking into account such independencies, as afforded by the factored MDP framework, can allow for compactly representing value functions and policies, and decreasing the sample complexity for learning them. We adapted the framework for intrinsically motivated skill learning in

MDPs discussed above to FMDPs by extending existing work on hierarchical decomposition of FMDPs and active learning of their dynamical structure. We showed the benefits of leveraging environmental structure on such skill learning in more complex environments, and their computational advantages in learning solutions to ensembles of related tasks in a given environment after a period of developmental exploration.

7.1.3 Incremental Structure Learning in Continuous Factored MDPs

We developed a novel algorithm for online, incremental learning of transition models for factored MDPs with continuous, multi-dimensional state and action spaces. Through the use of incremental density estimation techniques and information-theoretic principles, our algorithm learns a factored model of the transition dynamics of a continuous FMDP online from a single, continuing trajectory of experience. This approach provides a first step towards applying our framework for intrinsically motivated skill learning to more challenging and interesting problems that cannot be formalized as finite FMDPs.

7.1.4 Temporal Difference Networks for Continuous Dynamical Systems

We presented a novel algorithm for online, incremental learning of TD network representations of partially observable dynamical systems with continuous observations and actions. We showed that our algorithm is capable of learning accurate and robust models of several noisy, continuous, partially observable dynamical systems. This approach provides a first step towards applying our framework for intrinsically motivated skill learning to more challenging and interesting problems whose environmental dynamics cannot be modeled as MDPs.

7.2 Future Work

There are several interesting avenues for future research based on the ideas presented in this thesis. First and foremost is taking the next steps in applying the principles of Chapters 3 and 4 to the development of algorithms for intrinsically motivated skill learning in continuous domains with structured representations (e.g., continuous FMDPs) and partial observability, such as those for which we developed incremental model-learning techniques in Chapters 5 and 6. The characteristics of these environments pose challenges to the specific implementation of such algorithms, but there is nothing fundamental about these characteristics that invalidate the application of the principles of self-directed active learning we address in this thesis.

Also critical to the broader application of the principles of active learning we focus on in this work is the problem of automating option discovery. Recall in the algorithms of Chapters 3 and 4 that we assume an agent has a predefined set of interesting states or regions of the state space which serve as subgoals for the options it will learn. These are specified by the system designer based on domain knowledge or long-term objectives. An interesting line of research is to automatically discover these subgoals such that they maximize an agent's ability to control its environment. This would require some form of domain-independent criterion for selecting subgoals. There has been some existing work on this problem in MDPs, using metrics such as graph connectivity in MDPs (Simsek, 2008), change point detection in continuous MDPS (Konidaris, 2011), and convergence of pre-defined sensorimotor controllers to stable configurations (Hart et al., 2008). Whether there is a single metric or criterion that has general purpose applicability to this problem is an important open question.

Another useful research path involves defining an intrinsic reward function that relaxes the exploration policy from the approach we take in this thesis—exhaustively trying all actions in all states. Our incorporation of linear function approximation and factored representations goes some way in generalizing intrinsic value to similar states or equivalent states, but there is certainly room for improvement. An algorithm that could identify regions of the state space that are inherently uninteresting or irrelevant with respect to some long-term objective would increase the efficiency of an agent's exploration behavior. Such an approach would have to employ metrics that could be used to discount large regions of the state space without visiting them and observing their dynamics.

Of course another promising direction for exploration is within the space of intrinsic reward functions. The reward functions we defined in this work are only one instance of one class of potential reward functions that motivate agents to learn hierarchies of skills. It is fruitful to consider other instances in this class, and other classes as well. Indeed, it is even possible to automate the search for such functions by defining some metric of life-long performance for an agent and optimizing in the space of such functions. Some preliminary work in this vein has been explored by others already (Lewis et al., 2010; Niekum et al., 2010), but there are many potential extensions and alternatives.

We demonstrated the performance of our algorithms in continuous domains using radial basis functions (RBFs), which are an intuitive but relatively unsophisticated choice of basis for function approximation. There is an enormous body of work involving basis selection that considers tradeoffs between complexity, generalization capabilities, and representational capacity, among other properties. We chose not to focus on optimizing this selection to avoid distracting the reader from the primary objectives of the algorithms, which are largely independent of the choice of function approximator. However, applying some of these alternative choices of basis functions to our algorithms may result in improved learning speed and ability to generalize more readily to unfamiliar tasks.

In particular, the recent successes of deep learning methods for representation learning (Bengio, 2009; LeCun et al., 2015; Mnih et al., 2015) are an excellent candidate for applying the model-based active learning principles we address in this work to the acquisition of skill hierarchies in high-dimensional, partially-observable domains. As of yet, deep learning techniques have not been applied to hierarchical reinforcement learning scenarios, but they have been shown to be successful as representations of value functions for flat RL methods, and their recurrent implementations are certainly capable of serving as forward models of environmental dynamics. Since the structure of the representations produced by these methods is inherently hierarchical, they are a natural fit for representing action hierarchies as well as perceptual hierarchies. We feel these methods are currently the most promising direction for the application of the principles of intrinsically motivated learning of skill hierarchies to challenging adaptive control problems like many in the field of robotics.

BIBLIOGRAPHY

Abbeel, P., Koller, D., & Ng, A. Y. (2006). Learning factor graphs in polynomial time and sample complexity. *Journal of Machine Learning Research*, *7*, 1743–1788.

Barto, A. G., Singh, S., & Chentanez, N. (2004). Intrinsically motivated learning of hierarchical collections of skills. *International Conference on Developmental Learning (ICDL)*. LaJolla, CA.

Bellman, R. (1957). *Dynamic programming*. Rand Corporation research study. Princeton University Press.

Bengio, Y. (2009). Learning Deep Architectures for AI. *Foundations and Trends in Machine Learning*, *2*, 1–127.

Boutilier, C., Dearden, R., & Goldszmidt, M. (2000). Stochastic dynamic programming with factored representations. *Artificial Intelligence*, *121*, 49–107.

Brafman, R. I., & Tennenholtz, M. (2003). R-Max - a general polynomial time algorithm for near-optimal reinforcement learning. *Journal of Machine Learning Research*, *3*, 213–231.

Brown, A. L., & Kane, M. J. (1988). Preschool children can learn to transfer: Learning to learn and learning from example. *Cognitive Psychology*, *20*, 493–523.

Chickering, D., Geiger, D., & Heckerman, D. (1995). Learning bayesian networks: Search methods and experimental results. *Artificial Intelligence and Statistics (AISTATS)* (pp. 112–128).

Cohn, D., Atlas, L., & Ladner, R. (1994). Improving generalization with active learning. *Machine Learning*, *15*, 201–221.

Dean, T., & Kanazawa, K. (1989). A model for reasoning about persistence and causation. *Computational Intelligence*, *5*, 142–150.

Degris, T., Sigaud, O., & Wuillemin, P.-H. (2006). Learning the structure of factored Markov Decision Processes in reinforcement learning problems. *International Conference on Machine Learning (ICML)*. Pittsburgh, PA.

Dempster, A. P., Laird, N. M., & Rubin, D. B. (1977). Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society, Series B (Methodological)*, *39*, 1–38.

Diuk, C., Li, L., & Leffler, B. R. (2009). The adaptive k-meteorologists problem and its application to structure learning and feature selection in reinforcement learning. *International Conference on Machine Learning (ICML)* (pp. 249–256). New York, NY, USA: ACM.

Guestrin, C., Patrascu, R., & Schuurmans, D. (2002). Algorithm-directed exploration for model-based reinforcement learning in factored MDPs. *International Conference on Machine Learning (ICML)* (pp. 235–242).

Harlow, H. F., Harlow, M. K., & Meyer, D. R. (1950). Learning motivated by a manipulation drive. *Journal of Experimental Psychology*, *40*, 228–234.

Hart, S., Sen, S., & Grupen, R. (2008). Intrinsically motivated hierarchical manipulation. *IEEE Conference on Robots and Automation (ICRA)*. Pasadena, CA.

Jonsson, A., & Barto, A. G. (2006). Causal graph based decomposition of factored mdps. *Journal of Machine Learning Research*, *7*, 2259–2301.

Jonsson, A., & Barto, A. G. (2007). Active learning of dynamic Bayesian networks in Markov Decision Processes. *Symposium on Abstraction, Reformulation, and Approximation (SARA)*.

Kaplan, F., & Oudeyer, P.-Y. (2004). Maximizing learning progress: An internal reward system for development. In F. Idia, R. Pfeifer, L. Steels and Y. Kuniyoshi (Eds.), *Embodied artificial intelligence*, Lecture Notes in Artificial Intelligence (LNAI) 3139, 259–270. Springer-Verlag.

Kocsis, L., & Szepesvári, C. (2006). Bandit based monte-carlo planning. In *Machine learning: Ecml 2006*, 282–293. Springer.

Konidaris, G. D. (2011). *Autonomous robot skill acquisition.* Doctoral dissertation, University of Massachusetts Amherst.

Konidaris, G. D., & Barto, A. G. (2009). Efficient skill learning using abstraction selection. *International Joint Conference on Artificial Intelligence (IJCAI)*.

LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. *Nature*, *521*, 436–444.

Lewis, R. L., Singh, S., & Barto, A. G. (2010). Where Do Rewards Come From? *Proceedings of the International Symposium on AI-Inspired Biology* (pp. 2601–2606).

Li, L., Littman, M. L., & Walsh, T. J. (2008). Knows What It Knows: A framework for self-aware learning. *International Conference on Machine Learning (ICML)*.

Littman, M. L., Sutton, R. S., & Singh, S. (2002). Predictive representations of state. *In Advances In Neural Information Processing Systems (NIPS)* (pp. 1555–1561).

Mahadevan, S. (2008). *Representation discovery using harmonic analysis.* Morgan and Claypool.

Makino, T., & Takagi, T. (2008). On-line discovery of temporal-difference networks. *In International Conference on Machine Learning (ICML)*. Helsinki, Finland.

Menashe, J., & Stone, P. (2015). Monte Carlo Hierarchical Model Learning. *Proceedings of the 14th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2015)*.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., & Hassabis, D. (2015). Human-Level Control through Deep Reinforcement Learning. *Nature*, *518*, 529–533.

Mugan, J., & Kuipers, B. (2009). Autonomously learning an action hierarchy using a learned qualitative state representation. *International Joint Conference on Artificial Intelligence*.

Niekum, S., Barto, A. G., & Spector, L. (2010). Genetic Programming for Reward Function Search. *IEEE Transactions on Autonomous Mental Development*, *2*, 83–90.

Pan, S. J., & Yang, Q. (2010). A survey on transfer learning. *IEEE Transactions on Knowledge and Data Engineering*, *22*, 1345–1359.

Parr, R., Li, L., Taylor, G., Painter-Wakefield, C., & Littman, M. L. (2008). An analysis of linear models, linear value-function approximation, and feature selection for reinforcement learning. *International Conference on Machine Learning (ICML)*. Helsinki, Finland.

Rafols, E. J., Ring, M. B., Sutton, R. S., & Tanner, B. (2005). Using predictive representations to improve generalization in reinforcement learning. *In International Joint Conference on Artificial Intelligence (IJCAI)* (pp. 835–840).

Renyi, A. (1961). On measures of entropy and information. *Berkeley Symposium on Mathematical Statistics and Probability* (pp. 547–561). University of California Press.

Schmidhuber, J. (1991). A possibility for implementing curiosity and boredom in model-building neural controllers. *International Conference on Simulation of Adaptive Behavior (SAB): From Animals to Animats* (pp. 222–227). MIT Press/Bradford Books.

Schmidhuber, J. (2005). Self-motivated development through rewards for predictor errors/improvements. *American Association of Artificial Intelligence (AAAI)*.

Schwarz, G. (1978). Estimating the dimension of a model. *Ann. Statist.*, *6*, 461–464.

Simsek, O. (2008). *Behavioral building blocks for autonomous agents: Description, identification, and learning*. Doctoral dissertation.

Simsek, O., & Barto, A. G. (2006). An intrinsic reward mechanism for efficient exploration. *International Conference on Machine Learning (ICML).* Pittsburgh, PA.

Singh, S., & James, M. R. (2004). Predictive state representations: A new theory for modeling dynamical systems. *In Uncertainty in Artificial Intelligence (UAI)* (pp. 512–519).

Strehl, A. L., Diuk, C., & Littman, M. L. (2007). Efficient structure learning in factored-state MDPs. *American Association for Artificial Intelligence (AAAI).*

Sutton, R. S. (1988). Learning to predict by the methods of temporal differences. *Machine Learning* (pp. 9–44).

Sutton, R. S. (1991). Integrated modeling and control based on reinforcement learning and dynamic programming. *Advances in Neural Information Processing Systems (NIPS)* (pp. 471–478).

Sutton, R. S., & Barto, A. G. (1998). *Reinforcement learning: An introduction.* Cambridge, Massachusetts: MIT Press.

Sutton, R. S., Precup, D., & Singh, S. (1999). Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, *112*, 181–211.

Sutton, R. S., Rafols, E. J., & Koop, A. (2006). Temporal abstraction in temporal-difference networks. *In Advances in Neural Information Processing Systems (NIPS)* (pp. 1313–1320).

Sutton, R. S., & Tanner, B. (2005). Temporal-difference networks. *In Advances in Neural Information Processing Systems (NIPS)* (pp. 1377–1384).

Tanner, B., & Sutton, R. S. (2005). Td(λ) networks: temporal-difference networks with eligibility traces. *International Conference on Machine Learning (ICML)* (pp. 888–895).

Titterington, D. M., Smith, A. F. M., & Makov, U. E. (1985). *Statistical analysis of finite mixture distributions.* New York: Wiley.

Utgoff, P. E., Berkman, N. C., & Clouse, J. A. (1997). Decision tree induction based on efficient tree restructuring. *Machine Learning*, *29*, 5–44.

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

Vigorito, C. M. (2009). emporal-difference networks for dynamical systems with continuous observations and actions. *Proceedings of the Conference on Uncertainty in AI (UAI) '09.*

Wingate, D. (2008). *Exponential family predictive representations of state*. Doctoral dissertation, University of Michigan.

Wolfe, A. P., & Barto, A. G. (2006). Decision Tree Methods for Finding Reusable MDP Homomorphisms. *In Proceedings of AAAI-06* (pp. 530–535).

Wolfe, B., James, M. R., & Singh, S. (2008). Approximate predictive state representations. *In Autonomous Agents and Multiagent Systems (AAMAS)* (pp. 363–370). Estoril, Portugal.

Xiong, H., Swamy, M. N. S., Ahmad, M. O., & King, I. (2004). Branching competitive learning network: A novel self-creating model. *IEEE Transactions on Neural Networks*, *15*.

Yao, H., Szepesvári, C., Sutton, R., Modayil, J., & Bhatnagar, S. (2014). Universal Option Models. *Proceedings of the 28th Annual Conference on Neural Information Processing Systems (NIPS)* (pp. 1–9).

Yin, H., & Allinson, N. M. (2001). Self-organizing mixture networks for probability density estimation. *IEEE Transactions on Neural Networks*, *12*, 405–411.