Reinforcement Learning for Factored Markov Decision Processes

by

Brian Sallans

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy,
Graduate Department of Computer Science
in the University of Toronto

Abstract

Reinforcement Learning for
Factored Markov Decision Processes

Brian Sallans
Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto
2002

Learning to act optimally in a complex, dynamic and noisy environment is a hard problem. Various threads of research from reinforcement learning, animal conditioning, operations research, machine learning, statistics and optimal control are beginning to come together to offer solutions to this problem. I present a thesis in which novel algorithms are presented for learning the dynamics, learning the value function, and selecting good actions for Markov decision processes. The problems considered have high-dimensional factored state and action spaces, and are either fully or partially observable. The approach I take is to recognize similarities between the problems being solved in the reinforcement learning and graphical models literature, and to use and combine techniques from the two fields in novel ways.

In particular I present two new algorithms. First, the DBN algorithm learns a compact representation of the core process of a partially observable MDP. Because inference in the DBN is intractable, I use approximate inference to maintain the belief state. A belief-state action-value function is learned using reinforcement learning. I show that this DBN algorithm can solve POMDPs with very large state spaces and useful hidden state.

Second, the PoE algorithm learns an approximation to value functions over large factored state-action spaces. The algorithm approximates values as (negative) free energies in a product of experts model. The model parameters can be learned efficiently because inference is tractable in a product of experts. I show that good actions can be found even in large factored action spaces by the use of brief Gibbs sampling.

These two new algorithms take techniques from the machine learning community and apply them in new ways to reinforcement learning problems. Simulation results show that these new methods can be used to solve very large problems. The DBN method is used to solve a POMDP with a hidden state space and an observation space of size greater than 2^{180}. The DBN model of the core process has 2^{32} states represented as 32 binary variables. The PoE method is used to find actions in action spaces of size 2^{40}.

Acknowledgements

I am grateful to the many people who have contributed to this thesis, and to making my time in the Computer Science graduate program a great learning experience.

First I would like to thank my thesis advisor Geoffrey Hinton for his patience, guidance and seemingly boundless enthusiasm. His input and direction was invaluable in the creation of this thesis, and the research it contains. I would also like to thank the other members of my committee for their input and interest: Radford Neal, Zoubin Ghahramani and Craig Boutilier. All of my committee members made extensive and excellent comments on early work and drafts of the thesis, for which I am very grateful. I am also grateful to my examiners Brendan Frey and Mike Mozer, whose comments and questions helped to clarify the thesis.

Much of this work was completed while visiting the Gatsby Computational Neuroscience Unit, University College London. I particularly owe a great deal to Peter Dayan for his patience and comments. I would also like to thank Alberto Mendelzon and Vassos Hadzilacos who have encouraged and supported me during my time in Toronto.

Current and former members of the Neural Networks Research Group, University of Toronto, and the Gatsby Computational Neuroscience Unit have greatly contributed to this thesis, and to making work fun and stimulating: Hagai Attias, Andrew Brown, Brendan Frey, Quaid Morris, Alberto Paccanaro, Mike Revow, Sam Roweis, Maneesh Sahani, Yee Whye Teh, Emo Todorov and everyone else at Toronto and Gatsby. In particular, Andy, Alberto and Yee-Whye took the time to make valuable comments on initial drafts of this thesis. My research has also benefitted from the constant flow of visitors to the Neural Networks lab and to Gatsby.

My love and thanks go to my parents and brother Tim for their unwavering encouragement and support. And my love and thanks to Astrid who has helped give me the motivation to see this thesis through to completion.

I was financially supported by a Natural Sciences and Engineering Research Council of Canada post-graduate scholarship and by the Gatsby Charitable Foundation. My work has been financially supported by grants from the Natural Sciences and Engineering Research Council, the Institute for Robotics and Intelligent Systems, and by the Gatsby Charitable Foundation.

Contents

List of Tables

List of Figures

Nomenclature

In general, bold font indicates a multivariate quantity. Upper case bold letters are used for matrices, and lower case bold letters are used for vectors. Single elements of a vector are indexed by a subscript. Script capital letters indicate sets or fields. Sets are denoted with curly braces. Sets are either denoted by listing members: $\{x_1, x_2, ..., x_n\}$; or by use of subscripts: $\{x_i\}_{i=1}^{n}$. Vectors are sometimes written as elements in square brackets: $[a\ b]$ denotes a row vector, and $[a\ b]^\top$ denotes a column vector.

t	time index
s	single state variable
$\mathbf{s}$	multidimensional state variable
s_i	i^{th} element of multidimensional state variable
a	single action variable
$\mathbf{a}$	multidimensional action variable
a_j	j^{th} element of multidimensional action variable
o	single observation variable
$\mathbf{o}$	multidimensional observation variable
o_k	k^{th} element of multidimensional observation variable
$\mathbf{h}$	multidimensional hidden variable
h_i	i^{th} element of multidimensional hidden variable
ϕ	belief state
μ	factored approximate belief state
μ_i	i^{th} element of factored belief state
$\mathbf{W}$	Transition parameter (matrix) for DBN
$\mathbf{U}$	Emission parameter (matrix) for DBN
θ	generic parameter
α_θ	learning rate for parameter θ
r^t	immediate reward for time t
R^t	total reward from time t to end of task
$\pi(s, a)$	policy
$V^\pi(s)$	value function for policy π

$Q^{\pi}(s,a)$	action-value function for policy π
$V^{*}(s)$	optimal value function
$Q^{*}(s,a)$	optimal action-value function
E_B	Bellman error
E_B^T	TD error
E_B^S	TD error (SARSA)
E_B^Q	TD error (Q-learning)
$e^t(s,a)$	eligibility trace for state s, action a at time t
γ	discount factor
λ	TD parameter
$\delta(x)$	delta function
$\bigoplus$	cross-sum operator
$\bigcup$	set union
$\odot$	element-wise multiplication
$[\cdot]^{\top}$	matrix transpose

Chapter 1

Introduction

1.1 Motivation

When a learning agent is placed in a new environment for the first time it has no first-hand knowledge of the environment. It just has its innate preconceptions. As it explores the environment it modifies its preconceptions based on new experience. As it understands more about the world, the learner can start to make beneficial decisions.

As the agent moves through its environment it bases its actions on information from a number of sources. These include sensory input, efferent copies of motor commands and memories of previous inputs. The input is noisy and high-dimensional.

The actions that the agent takes at a given time have many attributes in common with the sensory input: They are high-dimensional and have an uncertain effect on the world. Possible actions might include motor commands, information-gathering, or even inactivity.

The agent must be able to deal with high-dimensional and uncertain states and actions in order to operate in a complex environment. Making inferences about the state of the world from noisy observations has been a popular subject of study in the artificial intelligence and machine learning communities. One formalism is the graphical model.

The purpose of a graphical model is to capture the distribution of observed data with a probabilistic model. The graphical representation of the model indicates which variables can be assumed to be conditionally independent. Given observations of some variables, inferring the distribution over unobserved (or hidden) variables is of paramount importance in using and learning the parameters of these models. Exact and approximate inference algorithms have been and still are intensely studied in the graphical models literature.

Acting on certain or uncertain information has been studied in a different body of literature. Reinforcement learning involves learning how to act so as to maximize a reward signal given samples of sequences of state-action pairs from the environment. It has been closely linked to Markov decision processes and stochastic dynamic programming. There has been work on reinforcement learning with high-dimensional states, state uncertainty and partial observability. In particular there are exact dynamic programming methods for solving fully and partially observable Markov decision processes. There are also approximate methods for dealing with real-valued state and action variables.

Perhaps surprisingly, there has been relatively little interaction between these two communities. There is a history of combining exact inference in graphical models with dynamic programming. These models are called influence diagrams. Combining approximate inference methods with approximate dynamic programming has been almost completely neglected. Influence diagrams solution techniques have been proposed where the problem of selecting good actions is reduced to the problem of doing inference in a Bayesian network. However, the combination of this transformation with approximate inference has received almost no attention.

At the most general level the purpose of this thesis is to combine methods made popular in these two distinct bodies of literature, and by doing so to derive novel algorithms that can solve previously unsolved classes of problems. In particular, I study combinations of approximate inference, approximate dynamic programming and model learning

that have not previously been considered. This includes combining algorithms in novel ways, and using algorithms from one domain to solve problems in another.

This thesis shows that several techniques from the graphical models literature can be fruitfully applied to the problem of learning to act in fully and partially observable Markov environments. I demonstrate that approximate inference and learning methods can be used both to learn about the dynamics of decision processes, and also to select actions by transforming the decision problems themselves into density estimation problems.

More specifically, I focus on high-dimensional, fully or partially observable factored Markov decision processes (factored (PO)MDPs). Factored (PO)MDPs are interesting because they have properties in common with large, interesting real-world problems. In particular, the state of the world is described in a factored MDP by a set of variables. This is a very natural and intuitive way to describe a complex environment. For example, we naturally describe the weather by talking about temperature, precipitation, wind, visibility and so on. At a much lower level, the information conveyed by our retinas is available as a set of neural activities that indicate how much light is hitting each photoreceptor. Each variable itself is simple and straightforward but the combination can make for a very detailed description of the world, and the relationship between variables can get very complicated.

Similarly the actions we can take in the world are well described by a set of values. At a low level we can describe our actions as a large set of muscle activations, or by a set of individual limb motions. At the opposite extreme we can describe our actions by a set of intended results: I want to get to work in the morning, stop by the post office, grab some lunch, and so on.

In this thesis I show that techniques from the graphical models literature can be fruitfully applied to reinforcement learning in factored MDPs. This is because density modeling techniques are designed to apply to exactly this kind of data. Density models often try to find the low-dimensional data subspace or manifold in a high-dimensional

input space. The models that I will discuss learn about the environment from experience, extract compact factored representations of the world, and allow us to make inferences about the world based on experience. In this thesis I show that factored models of states and actions can be learned by experience and used to select good actions.

In subsequent chapters I will discuss new ways in which inference and learning techniques can be applied to factored (PO)MDPs. For each of these new methods, I first briefly outline the problem to be solved. I then indicate the methods that will be used to solve it, and discuss simulation results. In the final chapter, I discuss bringing the techniques together and ways in which the methods might be extended.

1.2 Putting this work in context

The conjunction of approximate inference, model learning and reinforcement learning has received little attention. Mapping the problem of solving influence diagrams onto inference in Bayesian networks has been investigated by Shachter and Peot [1992] and Zhang [1998]. Although they suggest that approximate inference could be used, they do not pursue this possibility. Approximate inference was used to compute expectations and combined with traditional action selection techniques to solve influence diagrams by Charnes and Shenoy [1999] and **?**].

The combination of approximate inference with solution methods for POMDPs has been recently investigated by Rodriguez, Parr, and Koller [2000], Thrun [2000] and Poupart, Ortiz, and Boutilier [2001]. Poupart and Boutilier [2000] investigate tuning the approximate inference algorithm to explicitly maximize expected return. All of these methods assume knowledge of the dynamics of the process. In other words, these methods take as input a transition and emission distribution. Tadepalli and Ok [1996] learn the conditional probability tables of a dynamic Bayesian network which represents a factored MDP. They also learn a factored approximate value function. They do not consider par-

tial observability. In Chapter 3, I consider the case where a compact model of a POMDP is learned from experience. Approximate inference is used to maintain the belief state, and an approximate belief-state value function is learned and used to select good actions.

Value function representations for continuous action spaces are investigated by Baird and Klopf [1993], Doya [1996] and Santamaria, Sutton, and Ram [1998]. They each offer a different way of finding the optimal action in a continuous space of actions. Baird and Klopf [1993] is probably closest in flavour to the work presented in Chapter 4, in that they present a new value function approximation architecture for which the selection of good actions is particularly easy. Sabes and Jordan [1996] suggest a mapping from values of actions onto energies which is similar to the one used in Chapter 4. However, they confine themselves to immediate-reward problems.

The solution of factored (PO)MDPs is investigated in [Boutilier and Poole 1996; Dearden and Boutilier 1997; Schneider, Wong, Moore, and Riedmiller 1999; Boutilier, Dearden, and Goldszmidt 2000; Koller and Parr 2000; Peshkin, Kim, Meuleau, and Kaelbling 2000; Dearden 2001; Guestrin, Koller, and Parr 2002]. These include factored representations of states, actions and values. Some ([Boutilier and Poole 1996; Dearden and Boutilier 1997; Boutilier, Dearden, and Goldszmidt 2000]) consider the case when the actions can be explicitly enumerated and give an algorithm for finding the exact value function given a compact representation of an MDP. Others ([Dean, Givan, and Kim 1998; Schneider, Wong, Moore, and Riedmiller 1999; Koller and Parr 2000; Peshkin, Kim, Meuleau, and Kaelbling 2000; Guestrin, Koller, and Parr 2002]) present algorithms for finding approximate value functions and approximate policies given factored states and actions. They assume that the value function and policy can be expressed in a compact, factored form, where each component of the value function or policy depends on a small, pre-specified subset of state or action variables.

There has also been work on direct-policy and actor-critic methods for factored and continuous action spaces [Sutton, McAllester, Singh, and Mansour 2000; Ng, Parr, and

Koller 2000; Ng and Jordan 2000; Konda and Tsitsiklis 2000]. These methods are very interesting in the context of Chapter 4. When specific parameterized stochastic policies have been mentioned in previous work, they have been explicitly normalized. This restricts the family of available policies when using factored actions. We have to assume some independence between actions so that we only have to normalize over small subsets of action variables [Peshkin, Kim, Meuleau, and Kaelbling 2000; Guestrin, Koller, and Parr 2002]. This is in contrast to the method presented in Chapter 4, where a policy is implicitly specified by a parameterized energy function. As we will see in Chapter 4, it is possible to parameterize a policy implicitly without normalizing it, and sample actions according to the policy distribution using Markov chain Monte Carlo methods. This expands the class of policies that are available for use by direct policy methods. Direct policy methods are discussed further in section 5.4.1.

I am not aware of previous work which combines factored model learning, approximate inference and reinforcement learning for POMDPs. Nor am I aware of previous work on the representation of stochastic policies for large discrete action spaces where an explicit, normalized, compact parameterization of the policy is not available. Preliminary work originally appeared in [Sallans 2000] and [Sallans and Hinton 2001].

1.3 Contributions of this thesis

The thesis contributes:

1. A reference for researchers interested in the conjunction of machine learning, Bayesian inference and reinforcement learning.

2. A new approach to modeling factored POMDPs and factored value functions, which combines approximate inference, model estimation and factored value-function approximation.

3. A new technique for approximating value functions for large factored discrete and continuous MDPs and POMDPs.

4. A novel method for selecting actions based on this function approximator, even for very large action spaces.

5. A new representation for stochastic policies on large action spaces.

6. Matlab implementations of the decision problems studied in the thesis which can be used by future researchers.

1.4 Organization of this thesis

Chapter 2 covers background material necessary to understand the contributions made in subsequent chapters. Readers comfortable with these background topics can feel free to skip directly to chapter 3. Section 2.1 contains a formal definition of Markov decision processes. I then give an overview of methods for solving fully observable MDPs including dynamic programming and temporal difference learning. Section 2.2 extends this discussion to partially observable MDPs, and details a number of exact algorithms for solving POMDPs. I then discuss approximate algorithms for solving POMDPs. In section 2.3 I give a description of some graphical models that can be applied to MDPs and POMDPs including influence diagrams and dynamic Bayesian networks. I also discuss the advantages and challenges of using a factored representation of a (PO)MDP. I conclude the section with a short overview of inference and learning methods for graphical models.

Chapter 3 details a method for learning compact representations of the core process of a POMDP while simultaneously approximating the belief-state action-value function. Section 3.1 discusses factored state representation, and outlines the challenges that need to be overcome when learning to represent and act in a complex high-dimensional

POMDP. Sections 3.3–3.5 go into detail discussing the three main challenges: Learning, inference and value function estimation. Throughout this and subsequent sections I include an example architecture: The dynamic sigmoid belief network. In Section 3.6 I give the results of running a selection of algorithms on a set of successively harder tasks.

In Chapter 4 I present a method that allows for the compact representation of action-value functions for high-dimensional factored MDPs, and the efficient selection of good actions from a high-dimensional action space based on their estimated value. In sections 4.1 and 4.2 I discuss the problems unique to high-dimensional action spaces, and suggest a general method for approximating values and selecting actions. Section 4.4 and section 4.5 show specific examples of applying the method in the case of discrete and continuous action spaces respectively. In section 4.6 I detail the different update rules that can be used to learn the parameters of an approximate action-value function, and discuss the merits of the different updates. Section 4.7 gives the details of selecting good actions for high-dimensional action spaces. In section 4.8 I present simulation results on a number of discrete and continuous tasks.

In the final chapter, I discuss the two methods presented, and their relationship to one another. I summarize the contributions presented in this thesis. Finally I conclude with some discussion of the challenges still to be overcome and speculation on future trends in solving large (PO)MDPs.

Chapter 2

Background

Learning to act optimally in a complex, dynamic and noisy environment is a hard problem. Various threads of research from animal conditioning, operations research, machine learning, Bayesian statistics and optimal control are beginning to come together to offer solutions to this problem. In the following chapter I discuss some of these techniques and how they are related.

Specifically, I survey reinforcement learning in fully observable and partially observable Markov decision problems (section 2.1 and section 2.2), including definitions of the problems, and methods for finding exactly and approximately optimal solutions. I then discuss ways of representing Markov decision problems in efficient and compact ways (section 2.3). Finally, I outline inference and parameter learning in these compact parameterized models (section 2.4).

2.1 Markov Decision Processes

As an agent moves through its environment, it bases its actions on information received from a number of sources including sensory input, efferent copies of motor commands and memories of previous inputs. This information tells the learner something about the *state* of the world.

An agent in some state at time t executes an action and receives a reward from the environment. This is called a *decision process*. The task of learning which action to perform based on reward is called *reinforcement learning* [Sutton and Barto 1998]. If the next state is dependent on only the current state and action the decision process is said to obey the Markov property. This is called a Markov decision process (MDP) [Bellman 1957b] (see figure 2.1).

If the sets of states and actions are finite, then the problem is called a *finite* MDP. Much of the theoretical work done in reinforcement learning has focused on the finite case, and I will only discuss finite MDPs in this review. We can also distinguish between finite and indefinite horizon problems, where the task has a natural endpoint, and infinite horizon problems, where the task continues forever.

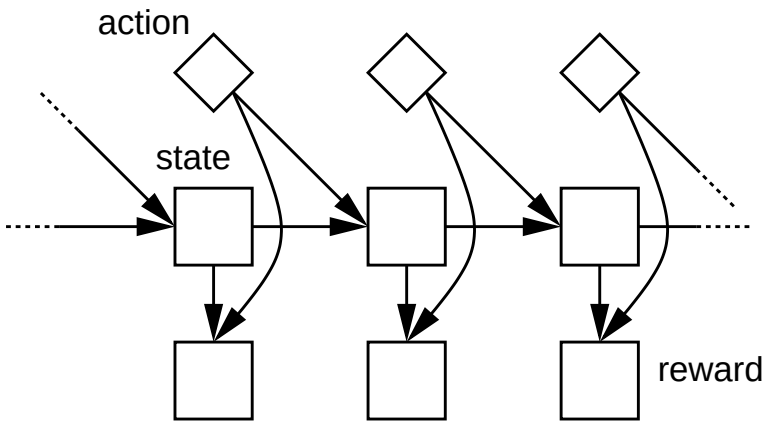

Figure 2.1: A Markov decision process. Squares indicate visible variables, and diamonds indicate actions. The state is dependent on the previous state and action, and the reward depends on the current state and action.

Formally, an MDP consists of:

- A set of states $\mathcal{S}$, and actions $\mathcal{A}$,
- A transition distribution $P(s^{t+1}|s^t, a^t)$, $s^t, s^{t+1} \in \mathcal{S}$, $a^t \in \mathcal{A}$, and
- A reward distribution $P(r^t|s^t, a^t)$, $s^t \in \mathcal{S}$, $r^t \in \mathbb{R}$, $a^t \in \mathcal{A}$.

In the above, t indexes the time step, which ranges over a discrete set of points in time. I will denote the transition probability $Pr(s^{t+1} = j \,|s^t = i, a^t = a)$ by $\mathbf{P}_{ij}(a)$. I will also denote the expected immediate reward received by executing action a in state i by $\mathbf{r}_i(a)$:

$$\mathbf{r}_i(a) = E\left\{r^t \,\middle|\, s^t = i, a^t = a\right\} \tag{2.1}$$

The goal of solving an MDP is to find a *policy* which maximizes the total expected reward received over the course of the task. A policy tells the learning agent what action to take for each possible state. It is a (possibly stochastic) mapping $\pi : \mathcal{S} \rightarrow \mathcal{A}$ from states to actions. The policy can be *non-stationary*, in which case a different mapping from states to actions can be used at each point in time. Alternatively, it can be *stationary*, in which case the same mapping is used at every point in time.

In the finite horizon case, the policy may be non-stationary. In the infinite horizon discounted case, there will exist an optimal stationary policy [Howard 1960]. From now on I will only focus on the infinite horizon discounted case, and stationary policies.

The expected *return* for a policy π is defined as the total reward that is expected when following policy π:

$$E_\pi\left\{R^t\right\} \quad = \quad E_\pi\left\{r^t + \gamma r^{t+1} + \gamma^2 r^{t+2} + ...\right\} \quad = \quad E_\pi\left\{\sum_{k=0}^{\infty} \gamma^k r^{t+k}\right\} \tag{2.2}$$

where t is the current time. Note that the discounting is required to ensure that the sum of infinite (discounted) rewards is finite, so that the quantity to be optimized is well-defined; the discount factor $\gamma \in [0, 1)$. Notice that the expectation is taken with respect to the policy π.

By assuming that the problem is Markov, we know that an optimal policy need only be a function of the current state; no other information is required to act optimally [Howard 1960].

The class of MDPs is a restricted but important class of problems. By assuming that

a problem is Markov, we can ignore the history of the process, and thereby prevent an exponential increase in the size of the domain of the policy. The Markov assumption underlies a large proportion of control theory, machine learning and signal processing including Kalman filters [Kalman 1960], hidden Markov models (HMMs) [Rabiner and Juang 1986], two-time-slice dynamic Bayesian networks [Dean and Kanazawa 1989], dynamic programming (DP) [Bellman 1957a] and temporal difference learning (TD) [Sutton 1988].

2.1.1 Dynamic Programming

Dynamic programming is a technique for finding solutions to optimization problems. It is similar to divide-and-conquer techniques, where a problem is broken down into subproblems that can be solved. To be amenable to dynamic programming, the optimization problem has to exhibit *optimal substructure*: An optimal solution must contain within it optimal solutions to subproblems.

In the case of MDPs, we must find a policy that produces the greatest expected return. With knowledge of transition probabilities $\mathbf{P}_{ij}(a)$ and expected immediate rewards $\mathbf{r}_i(a)$, and given a stochastic policy π, we can calculate the expected discounted return from the current state s:

$$\begin{aligned} V^\pi(s) &= E_\pi \left\{ \sum_{k=0}^{\infty} \gamma^k r^{t+k} \,\middle|\, s^t = s \right\} && (2.3) \\ &= E_\pi \left\{ r^t + \gamma \sum_{k=0}^{\infty} \gamma^k r^{t+k+1} \,\middle|\, s^t = s \right\} && (2.4) \\ &= \sum_a \pi(s,a) \sum_j \mathbf{P}_{sj}(a) \left[\mathbf{r}_s(a) + \gamma V^\pi(j) \right]. && (2.5) \end{aligned}$$

Here t denotes the current time. The function V^π is called the *value function* for policy π. The value function tells the agent the expected return that can be achieved by starting from any state, and then following policy π.

Eq.(2.5) is often called the Bellman equations for V^π. It is a set of $|\mathcal{S}|$ linear equations (one for each state $s \in \mathcal{S}$). The set of coupled equations can be solved for the unknown values $V^\pi(s)$. In particular, given some arbitrary initialization V_0^π, we can use the following iterative update:

$$V_{k+1}^\pi(s) = \sum_a \pi(s,a) \sum_j \mathbf{P}_{sj}(a) \left[\mathbf{r}_s(a) + \gamma V_k^\pi(j)\right]. \tag{2.6}$$

for all $s \in \mathcal{S}$. The iteration converges to the unique fixedpoint V^π as $k \to \infty$. This technique is called *iterative policy evaluation*. Each iteration is called a *full backup*: a *backup* because it backs up one step in the dynamics of the system by defining the value of a previous state in terms of the next state; and *full* because the value of every state is updated.

A policy is optimal if it assigns non-zero probability only to those actions that maximize the value function. The value function associated with an optimal policy π^* is called the optimal value function :

$$\begin{aligned} V^*(s) &= E_{\pi^*}\left\{\sum_{k=0}^{\infty} \gamma^k r^{t+k} \,\middle|\, s^t = s\right\} && (2.7)\\ &= \max_a E_{\pi^*}\left\{\sum_{k=0}^{\infty} \gamma^k r^{t+k} \,\middle|\, s^t = s, a^t = a\right\} && (2.8)\\ &= \max_a \sum_j \mathbf{P}_{sj}(a) \left[\mathbf{r}_s(a) + \gamma V^*(j)\right]. && (2.9) \end{aligned}$$

Again, Eq.(2.9) is actually $|\mathcal{S}|$ nonlinear equations, one for each state, and $|\mathcal{S}|$ unknowns, the values of the optimal value function. From Eq.(2.9) we can see that the problem of computing an optimal value function has optimal substructure.

There is an iterative DP algorithm, called *policy iteration* [Howard 1960], which is guaranteed to converge to the optimal value function. Given an arbitrary initial policy π^0 it iterates between two steps:

1. Compute the value function of the current policy V^{π^k}.

2. Improve the policy by greedily choosing actions which are optimal under V^{π^k}. Namely, let the new policy π^{k+1} be given by:

$$\pi^{k+1}(s) = \underset{a}{\operatorname{argmax}} \left\{ \sum_j \mathbf{P}_{sj}(a) \left[\mathbf{r}_s(a) + \gamma V^{\pi^k}(j) \right] \right\}. \tag{2.10}$$

The second step, called *policy improvement*, is guaranteed to improve the policy (if it is not already optimal), and the two steps, when iterated, will converge to an optimal policy and optimal value function. Note that the policy evaluation step is completed before the policy is improved. In particular, if an iterative policy evaluation algorithm is used, it is iterated to convergence.

As it turns out, it is unnecessary to allow the policy evaluation step to complete. The *value iteration* algorithm is similar to policy iteration, except that only a single full backup is performed in the first step. It can be shown that this algorithm also converges to an optimal policy.

If both algorithms result in an optimal policy, then which algorithm is better? While policy iteration requires complete policy evaluation between steps of policy improvement , this can usually be done in very few backups, because the value function typically changes very little when the policy is only slightly improved. At the same time, we would expect policy iteration to require fewer steps in total than value iteration (or at least no more) because it is making policy improvements based on more accurate information. In fact, this result has been shown theoretically [Putterman 1994]. However, which algorithm performs better for a given problem is an empirical question.

Instead of finding the value function $V^\pi(s)$, the *action-value* function $Q^\pi(s, a)$ can be found:

$$Q^\pi(s, a) \leftarrow \sum_j \mathbf{P}_{sj}(a) \left[\mathbf{r}_s(a) + \gamma \max_b Q^\pi(j, b) \right]. \tag{2.11}$$

Similarly, the optimal action-value function is denoted Q^*.

In the context of dynamic programming, action-value functions do not seem to make much sense. Instead of storing $|\mathcal{S}|$ parameters, we have to store $|\mathcal{S}||\mathcal{A}|$ ones. Action value functions are important for temporal difference learning, where we do not assume complete prior knowledge of the dynamics of the system.

2.1.2 Temporal Difference Learning

Temporal difference (TD) learning [Sutton 1988], like dynamic programming, addresses the reinforcement learning problem: how to act so as to maximize a reward signal provided by the environment. We can view TD as performing approximate dynamic programming to solve an MDP. The nature of the approximation and the speed with which TD algorithms converge have been the subjects of much study.

There are two major difficulties with applying dynamic programming to real-world problems. First, dynamic programming is centered around full backups. That is, the value of every state is updated in each iteration. Clearly this technique will not scale well to problems with many states. Second, DP requires that a perfect model of the environment be available. This is obviously not going to be the case for many problems.

Instead of using full backups, TD techniques use *sample backups*: They only update the value estimate for states as they are visited. The rationale is that although many states might be reachable, only a small subset will actually be visited in practice. For example, although the number of possible chess-board configurations is huge, the number seen in actual expert games is much smaller. There is no point in expending resources to update the value of states that will never be visited. This assumes that if a state has a reasonable likelihood of being visited after the value function is learned, it will be visited during estimation of the value function.

The distinction between DP and TD is that TD algorithms learn by interacting with their environment while DP requires a perfect model of the environment before it begins.

One possibility would be to learn a model of the environment by observing the results of performing actions, and then performing full or sample backups. Some algorithms do use this approach [Sutton 1990; Barto, Bradtke, and Singh 1995].

Backing up states only as they are visited gives rise to a number of TD update rules. Instead of using a known dynamics model to compute expected returns, these use samples of the dynamics found by interacting with the environment.

For example, the TD(0) update:

$$V(s^t) \leftarrow V(s^t) + \kappa \left[r^t + \gamma V(s^{t+1}) - V(s^t) \right] \tag{2.12}$$

where κ is a learning rate, uses samples from the environment to construct a Monte Carlo estimate of the expectations in Eq.(2.6). The update is a delta rule designed to move the estimated value function closer to the "bootstrapped" Monte Carlo estimate. If κ is reduced over time in the appropriate manner, and all states are visited an infinite number of times, then this algorithm will almost certainly converge to the value function of the current policy [Dayan 1992]. Since this method estimates the value function for the current policy, it can be incorporated into a policy iteration algorithm.

This update rule is called an *on-policy* update because it finds the value function for the policy which is currently being executed. The equivalent on-policy update rule for the action value function is known as SARSA [Rummery and Niranjan 1994; Sutton 1996]:

$$Q(s^t, a^t) \leftarrow Q(s^t, a^t) + \kappa \left[r^t + \gamma Q(s^{t+1}, a^{t+1}) - Q(s^t, a^t) \right]. \tag{2.13}$$

where κ is a learning rate. The name derives from the fact that the update depends on the set of values $\{s^t, a^t, r^t, s^{t+1}, a^{t+1}\}$.

While in the case of DP the difference between these two updates was small, now the difference is significant. If a state-value function is learned, then in order to extract a policy, a model of the environmental dynamics must also be learned. With SARSA,

however, this model is implicit in the action-value function, and a policy can be found simply by maximizing over possible actions in each state:

$$\begin{aligned} V^{\pi}(s) &= \max_{a} Q^{\pi}(s,a) \\ \pi(s) &= \operatorname*{argmax}_{a} Q^{\pi}(s,a) \end{aligned} \tag{2.14}$$

We will see that this difference becomes less important for partially-observable MDPs , because a model of the system dynamics must be learned in any case to maintain the belief state .

There is also an *off-policy* TD update rule for action value functions, called Q-learning [Watkins 1989; Watkins and Dayan 1992]:

$$Q(s^t, a^t) \leftarrow Q(s^t, a^t) + \lambda \left[r^t + \gamma \max_{a} Q(s^{t+1}, a) - Q(s^t, a^t) \right]. \tag{2.15}$$

In this case, the optimal value function is estimated directly. The policy used to sample from the states and rewards is irrelevant, provided that all states and actions continue to be visited.

This version of the update rule is referred to as Q(0). There are more general updates called TD(λ) and Q(λ) which make use of information farther back in time than just the previous time step [Sutton 1988]. The most extreme case is TD(1), which averages the rewards over entire trials of the task before updating the value estimates. This technique is also called a pure Monte Carlo technique, since it uses sampling of entire trials to update the values of the expected returns (see [Sutton and Barto 1998] for a discussion of Monte Carlo decision process algorithms).

2.2 Partially Observable Markov Decision Processes

Many interesting decision problems are not Markov in the observations. One way to solve such problems in an MDP framework is to construct extended states by explicitly recording the history of the process for an appropriate number of steps; in other words consider an n^{th} order Markov process. There is a difficulty in determining the order; i.e., how many steps should be included in the history to transform the problem into an MDP? One way to determine this is by testing the statistics of reward predictions based on the current states, and expanding the histories until the reward distributions are Markov in the extended states. This is the approach taken by the U-Tree algorithm [McCallum 1995]. The U-Tree algorithm is quite flexible, allowing the process to have different order in different parts of the state space, and taking advantage of a factored input representation if it is provided. It cannot learn its own factored representation of the input, however.

Another approach is to assume that the process is Markov in some state variables that can not be observed. The observed variables (and rewards) at time t are then assumed to be conditionally dependent on the hidden state at time t and, given this hidden state, independent of all other hidden and visible variables (see figure 2.2).[1] The unobserved Markov process is called the *core* process.

This second approach can be formalized as a partially observable Markov decision process (POMDP). A POMDP consists of:

- A set of core states $\mathcal{S}$, a set of actions $\mathcal{A}$, and a set of observables $\mathcal{O}$,
- An initial distribution over core states $\phi^0 \in \Delta(\mathcal{S})$,

[1] More generally, we can assume that the observations and rewards are conditioned on the current state, current action and next state. However, this is equivalent to conditioning on just the current state and action for an expanded core state space.

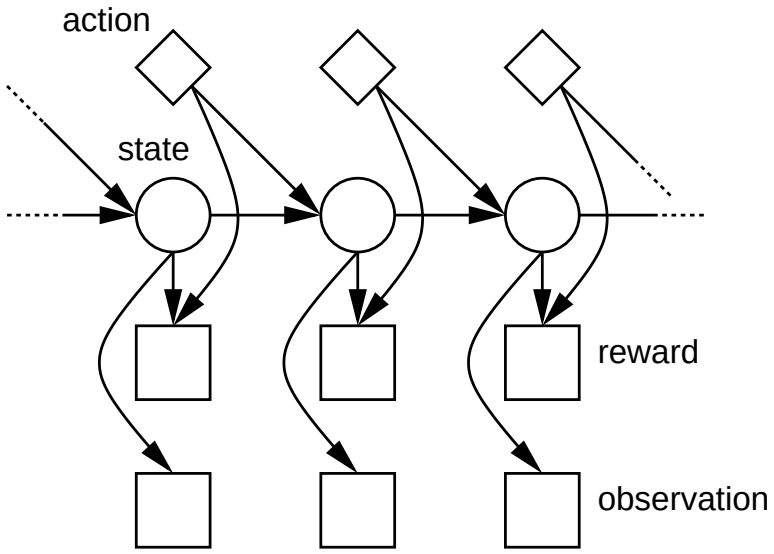

Figure 2.2: A partially observable Markov decision process. Circles indicate hidden variables, squares indicate visible variables, and diamonds indicate actions. The hidden state is dependent on the previous state and action, and the reward depends on the current state and action. The observation is just dependent on the current state.

- A transition distribution $P(s^{t+1}|s^t, a^t)$, $s^t, s^{t+1} \in \mathcal{S}$, $a^t \in \mathcal{A}$,
- A reward distribution $P(r^t|s^t, a^t)$, $s^t \in \mathcal{S}$, $r^t \in \mathbb{R}$, $a^t \in \mathcal{A}$, and
- An output distribution $P(o^t|s^t)$, $s^t \in \mathcal{S}$, $o^t \in \mathcal{O}$.

where $\Delta(\mathcal{S})$ denotes the simplex of dimensionality $|\mathcal{S}|$.

With a POMDP, we assume that a problem is Markov, but with respect to some *unobserved* random variable. The hidden variable summarizes all of the relevant information about the history of the observations. The state of the core variable at time t, denoted s^t, is dependent only on the state at the previous time step and on the action performed. The currently observed evidence, denoted o^t, is assumed to be independent of all other states and observations given the current state.

The state of the hidden variable is not known with certainty. A distribution over states, called the *belief state*, is maintained instead. It turns out that the process is also Markov in the belief state. The belief state transforms a discrete, hidden MDP into a fully-observable MDP whose states lie in the simplex $\Delta(\mathcal{S})$.

At each time step, the beliefs are updated by using Bayes' theorem to combine the belief state at the previous time step with newly observed evidence. For some evidence $o^t = j$ and action $a^t = a$, this is written:

$$\phi^{t+1} = T(\phi^t, j, a) = \frac{1}{\mathcal{Z}}\mathbf{O}^j\mathbf{P}(a)\phi^t \tag{2.16}$$

where $[\mathbf{P}(a)]_{ji} = P(s^{t+1} = j|s^t = i, a^t = a)$, $\mathbf{O}^j$ is a diagonal matrix with $[\mathbf{O}^j]_{ii} = P(o^t = j|s^t = i)$, and $\mathcal{Z}$ is a normalizing constant.

In the case of discrete time and finite discrete states, actions and observables (called a *finite* POMDP) a POMDP corresponds to a hidden Markov model (HMM) with a distinct transition matrix for each action. Exact belief updates for a given sequence of actions and observations can be computed tractably for this model with a forward pass of the forward-backward algorithm [Rabiner and Juang 1986].

One way to find an optimal policy is to learn a value function. Because the process is Markov in the belief state, the domain of the value function is $\Delta(\mathcal{S})$.

Sondik [1976] showed that for finite-horizon POMDPs the value function is piecewise linear and convex in the belief state. In the worst case, the number of linear pieces grows exponentially with the distance to the horizon, making exact computation of the optimal value function intractable in many cases. In the infinite horizon limit, the function may no longer be piecewise linear. However, there is a subclass of infinite-horizon POMDPs, called *finitely transient*, for which the value function is still piecewise linear [Sondik 1973]. In fact, even if the POMDP is not finitely transient, its optimal value function can be approximated to an arbitrary degree of accuracy by a piecewise linear and convex function. The computation (exact and approximate) of piecewise linear value functions has therefore played a central role in the study of POMDPs.

The POMDP formalism suffers from a similar difficulty as the explicit n^{th} order Markov representation: How do we know how many hidden states are enough to make

the core process first-order Markov? This problem has been addressed both in the context of HMMs [Stolcke and Omohundro 1993] and POMDPs represented as HMMs [Chrisman 1992; McCallum 1995].

Notice that even if the correct number of hidden states is found, the localist representation of a POMDP is exponentially inefficient: Encoding n bits of information about the history of the process requires 2^n states. This does not bode well for the abilities of localist models to scale up to problems with high-dimensional inputs and complex non-Markov structure. Instead of representing POMDPs as HMMs, they can be represented more compactly as dynamic Bayesian networks . See for example Dean and Kanazawa [1989], Russell [1998] or Boutilier and Poole [1996]. I will discuss this further in section 2.3.2.

2.2.1 Exact Algorithms

There have been a number of algorithms proposed to compute the optimal value function of a POMDP [Lovejoy 1991]. Most exact algorithms for computing the optimal value function for a finite-horizon POMDP use a DP step to compute the current value function from a previous value function (as in Eq.(2.9)). I will present five value-iteration algorithms: Monahan's exhaustive enumeration algorithm [Monahan 1982], incremental pruning [Zhang and Liu 1996; Cassandra, Littman, and Zhang 1997], Sondik's one-pass and two-pass algorithms [Sondik 1976; Smallwood and Sondik 1973] and the Witness algorithm [Littman, Cassandra, and Kaelbling 1996].

All of the algorithms begin by finding the optimal value function for a POMDP with horizon one (i.e., where a single action is performed), and in the k^{th} step use the optimal value function for the horizon $k-1$ problem to compute the optimal value function for a length k POMDP.

Sondik [1976] showed that the optimal value function for a length k finite-horizon POMDP is piecewise linear and convex. It can be represented by a unique minimal set

of vectors $\mathbf{A}^k$ such that

$$V_k^*(\phi) = \max_{\alpha \in \mathbf{A}^k} \left\{ \alpha^\top \phi \right\}. \tag{2.17}$$

Because each stage builds on the last, computing the unique *minimal* representation at each stage is important in reducing overall computation. All of the algorithms try to compute the unique minimal representation.[2]

The five algorithms can be broken down into two general categories: The first category computes a next-stage value function by first computing a large set of vectors and pruning away the excess, leaving the minimal set. The exhaustive and incremental pruning algorithms operate in this way. The second category, Sondik's algorithms and the Witness algorithm, build up the minimal set of vectors one by one by starting with a single vector, and then considering only "nearby" vectors.

Exhaustive Enumeration

Monahan introduced an exhaustive enumeration algorithm for finite-horizon POMDPs under the name "Sondik's one-pass algorithm" [Monahan 1982]. We can represent the value function for a finite-horizon POMDP of length k by a finite set of vectors $\mathbf{A}^k$. The algorithm first computes the optimal value function for a one-step process and works backwards. After the k^{th} step, the optimal value function has been computed for an k-step POMDP.

The first set of vectors are just the immediate rewards for each action:

$$\mathbf{A}^1 = \{\mathbf{r}(a) : a \in \mathcal{A}\}. \tag{2.18}$$

where $\mathbf{r}(a)$ is an action-specific vector giving the expected immediate reward for each core state.

[2] Although Sondik's algorithms are not guaranteed to do so in the worst case.

The k^{th} set of *candidate* vectors is given by:

$$\overline{\mathbf{A}^k} = \left\{ \alpha : \alpha = \mathbf{r}(a) + \sum_j \left[\mathbf{O}^j \mathbf{P}(a)\right]^\top \alpha_j, a \in \mathcal{A}, \alpha_j \in \mathbf{A}^{k-1} \right\} \tag{2.19}$$

where $\mathbf{P}(a)$ and $\mathbf{O}^j$ are defined previously (see Eq.(2.16)). Notice that the new value function is composed of two parts: the immediate reward and a weighted sum of values from the next stage value function. This weighted sum is computed for all possible assignments of vectors in the previous set $\mathbf{A}^{k-1}$ to possible observations.

After the new set of candidates is computed, it is pruned of dominated vectors by solving a linear programming problem for each vector $\alpha \in \overline{\mathbf{A}^k}$:

$$\max_{\phi} \left\{ \delta : [\alpha - \alpha']^\top \phi \leq \delta, \alpha' \in \overline{\mathbf{A}^k} \backslash \{\alpha\}, \delta \in \mathbb{R} \right\} \tag{2.20}$$

If the solution is such that $\delta > 0$ then there is some belief state for which α dominates all other vectors in $\overline{\mathbf{A}^k}$. Otherwise, α should not be included in $\mathbf{A}^k$. Following Cassandra, Littman, and Zhang [1997], I denote this pruning step by $\text{prune}(\cdot)$.

While simple in principle, this is clearly an expensive procedure. The number of vectors in the new set (prior to pruning) scales exponentially with the number of possible values an observation might produce, and we require a linear program for each vector. In practice, this procedure is too expensive for all but the smallest problems [Cassandra, Littman, and Zhang 1997].

Incremental Pruning

The incremental pruning algorithm [Zhang and Liu 1996; Cassandra, Littman, and Zhang 1997] is conceptually almost as simple as exhaustive enumeration. Following Cassandra, Littman, and Zhang [1997], I denote the *cross-sum* operation by $\bigoplus$. This is a set-construction operator defined by $\bigoplus_i \mathbf{A}_i = \{\sum_i \alpha_i : \alpha_i \in \mathbf{A}_i\}$. It is the sum of all possible combinations of the elements of the sets $\mathbf{A}_i$.

Let:

$$\mathbf{A}_{a,j}^k = \left\{ \frac{1}{|\mathcal{O}|}\mathbf{r}(a) + [\mathbf{O}^j \mathbf{P}(a)]^\top \alpha_j : \alpha_j \in \mathbf{A}^{k-1} \right\}. \tag{2.21}$$

Using this notation, the previous algorithm can be written simply as:

$$\begin{aligned} \mathbf{A}^1 &= \text{prune}\left(\{\mathbf{r}(a)\}\right) \\ \mathbf{A}^k &= \text{prune}\left(\bigcup_a \bigoplus_j \mathbf{A}_{a,j}^k\right) \end{aligned} \tag{2.22}$$

The incremental pruning algorithm hinges on the fact that:

$$\text{prune}\left(\mathbf{A} \bigoplus \mathbf{B} \bigoplus \mathbf{C}\right) = \text{prune}\left(\text{prune}\left(\mathbf{A} \bigoplus \mathbf{B}\right) \bigoplus \mathbf{C}\right) \tag{2.23}$$

The incremental pruning algorithm can be stated simply as:

$$\mathbf{A}^1 = \text{prune}\left(\{\mathbf{r}(a)\}\right)$$

$$\mathbf{A}_a^k = \text{prune}\left(\ldots\left(\text{prune}\left(\mathbf{A}_{a,1}^k \bigoplus \mathbf{A}_{a,2}^k\right) \bigoplus \mathbf{A}_{a,3}^k\right) \bigoplus \ldots\right) \tag{2.24}$$

$$\mathbf{A}^k = \text{prune}\left(\bigcup_{a \in \mathcal{A}} \mathbf{A}_a^k\right) \tag{2.25}$$

Cassandra, Littman, and Zhang [1997] showed that the number of linear programs required by this algorithm scales only linearly with the number of observations in the worst case, in contrast to the exponential scaling of the previous algorithm. In practice, incremental pruning is currently one of the fastest exact POMDP solution algorithms [Cassandra 1998].

Sondik's Algorithms

Sondik was the first to publish exact algorithms for finite and infinite-horizon POMDPs [Sondik 1976; Smallwood and Sondik 1973; Sondik 1973]. His algorithms, like the previous ones, involve setting up and solving a series of linear programming problems to

disqualify candidate vectors from the set $\mathbf{A}^k$. The difference is in how these candidate vectors are chosen.

The "two-pass" algorithm[3] consists of the same two steps as incremental pruning (Eq.(2.24) and Eq.(2.25)): In the first "pass" the sets $\mathbf{A}_a^k$ are computed, and in the second pass these sets are merged into the final set $\mathbf{A}^k$. The difference is in how the sets $\mathbf{A}_a^k$ are found.

Definition : The *witness region* of a vector $\alpha \in \mathbf{A}_a^k$ is defined as the set of all belief states for which α dominates all other vectors in $\mathbf{A}_a^k$:

$$R(\alpha, \mathbf{A}_a^k) = \left\{\phi : \alpha^\top \phi \geq \beta^\top \phi, \beta \in \mathbf{A}_a^k \backslash \{\alpha\}\right\} \tag{2.26}$$

Definition : Recall from the exhaustive enumeration algorithm (section 2.2.1) that the set of all candidate vectors $\overline{\mathbf{A}_a^k}$ is generated by considering all choices of α_j for each $j \in \mathcal{O}$. Given a vector $\alpha \in \overline{\mathbf{A}_a^k}$, the *neighbours* of α, denoted $\mathcal{N}(\alpha)$ are all of the vectors that differ from α by only one of these choices. So, if $\{\alpha_j\}_{j=1}^{|\mathcal{O}|}$ denotes a choice of $|\mathcal{O}|$ vectors (with replacement) from $\mathbf{A}_a^{k-1}$, and:

$$\alpha = \mathbf{r}(a) + \sum_j \left[\mathbf{O}^j \mathbf{P}(a)\right]^\top \alpha_j \tag{2.27}$$

then $\mathcal{N}(\alpha)$ is given by:

$$\begin{aligned} \mathcal{N}(\alpha) \;=\; \Bigg\{ & \alpha' : \exists k \; \alpha' = \mathbf{r}(a) + \sum_{j \neq k} \left[\mathbf{O}^j \mathbf{P}(a)\right]^\top \alpha_j + \\ & \left[\mathbf{O}^k \mathbf{P}(a)\right]^\top \beta, \beta \in \mathbf{A}_a^{k-1} \backslash \{\alpha_k\} \Bigg\} \end{aligned} \tag{2.28}$$

The two-pass algorithm will make use of the following two facts:

[3]Sondik refers to this as "the one-pass algorithm applied to the single action problem", but I will follow the terminology of Cassandra [1998] in calling it the "two-pass" algorithm.

Fact 1 : Although it is difficult to compute the set of all vectors required to represent the optimal value function, it is easy to compute the single vector from this set that is "active" for a *particular* belief state, action and observation. Given a belief state ϕ^t, an action a, and an observation j, the active vector is given by:

$$\alpha(\phi^t, a, j) = \frac{1}{|\mathcal{O}|}\mathbf{r}(a) + \left[\mathbf{O}^j\mathbf{P}(a)\right]^\top \alpha_{k-1}(T(\phi^t, j, a)) \tag{2.29}$$

where $\alpha_{k-1}(\phi) = \text{argmax}_\alpha \left\{\alpha^\top\phi\right\}$, $\alpha \in \mathbf{A}^{k-1}$. The active vector irrespective of observation $\alpha(\phi^t, a) = \sum_j \alpha(\phi^t, a, j)$.

Fact 2 (The Neighbour Theorem [Cassandra 1998]): For any $\alpha \in \overline{\mathbf{A}_a^k}$, there exists a belief state ϕ and a $\beta \in \overline{\mathbf{A}_a^k}$ such that

$$\phi^\top\beta > \phi^\top\alpha \tag{2.30}$$

if and only if there exists a $\nu \in \mathcal{N}(\alpha)$ where

$$\phi^\top\nu > \phi^\top\alpha \tag{2.31}$$

In other words, if there is some candidate vectors that dominates α at a belief state, then there is a neighbour that dominates it at that belief state.

Here are the basic steps of the two-pass algorithm:

1. Begin with some vector $\alpha \in \mathbf{A}_a^k$.
2. Find witness regions that border on the witness region of α (see step 2 below), and vectors that are active over these bordering regions.
3. Add these vectors to $\mathbf{A}_a^k$.
4. Iterate until all regions have been explored, and so all vectors in $\mathbf{A}_a^k$ have been

found.

5. When all sets $\mathbf{A}_a^k$ have been found, compute $\mathbf{A}^k = \text{prune}\left(\bigcup_a \mathbf{A}_a^k\right)$.

Step 1 : Fact 1 tells us that the first step of the algorithm is feasible: Let $\mathbf{A}_a^1 = \{\alpha(\phi, a)\}$ for some arbitrary belief state ϕ.

Step 2 : Step 2 is performed by first finding all of the neighbours α' of α, and then determining whether or not the witness regions of the neighbours border on the witness region of α. This is done by solving the LP problem:

$$\max_{\phi} \left\{ \delta : [\alpha' - \alpha]^\top \phi \leq \delta, \alpha' \in \mathcal{N}, \phi \in R(\alpha, \mathbf{A}_a^k), \delta \in \mathbb{R} \right\} \tag{2.32}$$

If the linear program has a solution such that $\delta = 0$, then $R(\beta, \mathbf{A}_a^k)$ borders $R(\alpha, \mathbf{A}_a^k)$. We know that searching through neighbours is sufficient to find all bordering regions because of the neighbour theorem.

Step 3 : Step 3 is simply performed by adding to $\mathbf{A}_a^k$ any of the neighbours that have bordering witness regions.

Step 4 : Set $\alpha \leftarrow \nu$ for some $\nu \in \mathbf{A}_a^k$ whose neighbours have not yet been tested, and repeat steps 1-3.

Unfortunately, a candidate neighbour may only be active along the boundary region. If this is true for some neighbours then the two-pass algorithm will still return a set of vectors that represents V_a^k, but it will not give the minimal set. In fact, it may yield a set that is exponentially larger (in the number of observations) than the minimal set. Its worst-case running time is exponential in $|\mathcal{O}|$, although its best-case running time is polynomial in $|\mathcal{O}|$. The Witness algorithm [Littman, Cassandra, and Kaelbling 1996] was designed to overcome this problem.

In practice, the two-pass algorithm seems to perform better than the Witness algorithm, and as well as incremental pruning, on a large range of problems [Cassandra 1998]. It seems that the problem of returning large numbers of unnecessary vectors is mostly a theoretical concern.

The "one-pass" algorithm [Sondik 1976; Smallwood and Sondik 1973] follows the same procedure as the two-pass algorithm, building up a set of vectors by exploring witness regions. It tries to construct the set $\mathbf{A}^k$ directly, without first constructing the sets $\mathbf{A}_a^k$. This avoids the second "merging" step. Because the algorithm must take into account the active vectors for all actions, the constraints used by the one-pass algorithm in its LP problems are more complicated than in the two-pass case. The regions explored turn out to be subsets of the actual witness regions of the active vectors. There are a number of issues involved in the implementation of the algorithm, and for the sake of brevity, I will not discuss them here. Although the one-pass algorithm is more complicated to implement, and has worse performance in the worst case than more recent algorithms, it deserves credit as the first exact algorithm for solving POMDPs .

The Witness Algorithm

The witness algorithm also builds the sets $\mathbf{A}_a^k$ by exploring regions in belief space, and then combining them to find $\mathbf{A}^k$. It also proceeds by looking at the neighbours of vectors already known to be in $\mathbf{A}_a^k$. However, instead of trying to determine if the new vector's witness region borders on a region already explored, the witness algorithm just finds a single belief state at which the new vector dominates the vectors already known to be in $\mathbf{A}_a^k$. This belief state, if found, is called a "witness" for the new vector. By the neighbour theorem, this strategy will eventually find such a belief state for some neighbour if $\mathbf{A}_a^k$ is not yet complete. A witness can be found (if it exists) by solving a LP like that in Eq.(2.20).

The steps of the witness algorithm are:

1. Compute the dominating vector α for an arbitrary belief state ϕ. Add α to $\mathbf{A}_a^k$. Initialize the set of neighbours to be investigated $\mathcal{N} = \emptyset$.

2. Find the neighbours of α, and add them to $\mathcal{N}$.

3. Remove a neighbour β from $\mathcal{N}$, and try to find a witness for β.

4. If a witness ϕ exists: Compute the dominating vector α at ϕ; add it to $\mathbf{A}_a^k$; and if $\alpha \neq \beta$, add β back to $\mathcal{N}$.

5. repeat steps 2-4 until $\mathcal{N}$ is empty.

The witness algorithm avoids the problems of the two-pass algorithm by using witnesses instead of looking for bordering regions, and it is conceptually quite simple. Its worst-case running time is polynomial in $|\mathcal{O}|$ and $|\mathbf{A}_a^{k-1}|$. In terms of empirical performance, however, it actually does worse than the two-pass algorithm and incremental pruning [Cassandra, Littman, and Zhang 1997].

2.2.2 Approximation Algorithms

Approximation algorithms have a long history of being applied to RL tasks for MDPs. Examples include feed-forward multilayer perceptrons [Bertsekas and Tsitsiklis 1996] and radial basis functions [Sato and Ishii 1999]. They are used both in continuous state or action domains, and to aggregate groups of discrete states in the case where the number of states is too large to use table-lookup.

There have not been as many attempts to use function approximators to learn the value functions of POMDPs . However, since a POMDP is equivalent to an MDP where the states lie on a simplex, much of the previous work on real-valued MDPs is relevant. I focus here on two basic techniques: early termination of an exact algorithm, and parameterized function approximation. There are other techniques such as grid-based interpolation [Lovejoy 1991], which operate by maintaining value function estimates on

a grid of points, and estimating values of other points by interpolation. I will not discuss these methods in detail here.

Early-termination

One obvious approximation technique for an infinite-horizon POMDP is to simply terminate an exact algorithm before it finishes. Sondik provided bounds on the error that can be expected from such a procedure in terms of the discount factor γ and the number of stages k [Sondik 1973]:

$$\|V^*(\cdot) - V_k^*(\cdot)\|_\infty \leq \left[\frac{\gamma^k}{1-\gamma^k}\right] \frac{K}{1-\gamma} \tag{2.33}$$

where $K = \max_{i,a} \mathbf{r}_i(a) - \min_{i,a} \mathbf{r}_i(a)$.

The performance of approximations similar to this one have been compared to a number of other approximations. In general early termination seems to compare favorably, although there are particular problems on which it does very poorly [Cassandra 1998].

Function Approximation

Another approach is to use a function approximator to learn an approximate value function. If the approximation is differentiable, its parameters can be learned by applying the delta rule to the Bellman error:

$$E_B(\phi) = \left[\mathbf{r}(a)^\top \phi + \gamma \max_a \sum_j V(T(\phi, j, a))\mathbf{O}^j \mathbf{P}(a)\phi\right] - V(\phi) \tag{2.34}$$

or to the TD error:

$$E_B^T(\phi^t) = \left[r^t + \gamma V(\phi^{t+1})\right] - V(\phi^t) \tag{2.35}$$

In terms of a parameter θ of the value function $V(\phi;\theta)$, the update rule is:

$$\Delta\theta = \lambda E_B(\phi)\nabla_\theta V(\phi;\theta) \tag{2.36}$$

where λ is a learning rate.

This is the approach taken by Bertsekas and Tsitsiklis [Bertsekas and Tsitsiklis 1996], Littman et al. [Littman, Cassandra, and Kaelbling 1995] and Parr and Russell [Parr and Russell 1995], among others. Littman et al. use a linear approximation to the action value function:

$$Q(\phi, a) = \mathbf{q}_a{}^\top \phi \tag{2.37}$$

Parr and Russell reason that, since the actual value function can be represented by maximizing over a set of dot-products:

$$\max_\alpha \left\{\alpha^\top \phi\right\}, \tag{2.38}$$

then a reasonable differentiable approximation would be the use of a softmax function over a set of dot products:

$$V(\phi) = \sqrt[k_m]{\sum_{\upsilon\in\boldsymbol{\Upsilon}} \left(\upsilon^\top\phi\right)^{k_m}} \tag{2.39}$$

where the number of vectors in $\boldsymbol{\Upsilon}$ is chosen arbitrarily. Typically, the parameter k_m is initially set to make the function quite smooth. It is then reduced during learning so that the approximated value function approaches a piecewise-linear function. Bertsekas and Tsitsiklis use a standard feedforward multilayer perceptron trained as in Eq.(2.36) where the derivative of the network is computed with backpropagation. They apply their technique to real-valued MDPs. The same approach has been applied to belief state MDPs [Rodriguez, Parr, and Koller 2000].

Results with these approaches are mixed. Parr and Russell report good initial results on small tasks [Parr and Russell 1995]. Recently their technique has been applied to

larger problems [Rodriguez, Parr, and Koller 2000]. The results are comparable to using a neural network function approximator with backpropagation. Littman et al. report mixed results for the linear approximator. On some tasks the simple linear approximation does surprisingly well, but on other tasks it does quite poorly [Littman, Cassandra, and Kaelbling 1995]. Note that none of these techniques are guaranteed to converge. At best, the linear approximation has been shown to converge to a bounded region for the SARSA update rule [Gordon 2001].

In chapter 3 I compare the results of several methods for belief-state MDPs, and show results for versions of the function approximators suitable for factored state representations.

2.3 Generative Models for Control

Simply put, a generative model is a statistical model which, when trained on a particular data set, can generate new data points. The goal is to duplicate the probability distribution from which the original data were drawn, so that newly generated points are statistically similar to those on which the model was trained. A hidden Markov model is an example of a generative model. In this section I discuss two other generative models that can be applied to control problems: influence diagrams and dynamic Bayesian networks . Unlike the HMM, they are not exponentially inefficient in their state representation.

2.3.1 Influence Diagrams

Influence diagrams are like Bayesian networks, but with three types of nodes: chance nodes, action nodes, and a single value node (see figure 2.3) [Howard and Matheson 1984]. The nodes represent random variables, actions, and the value of a utility function. Edges represent dependencies between variables. Chance nodes can be either unobserved

(hidden nodes) or observed (evidence nodes). The agent sets the values of action nodes, which then influence the distributions or value of other nodes.

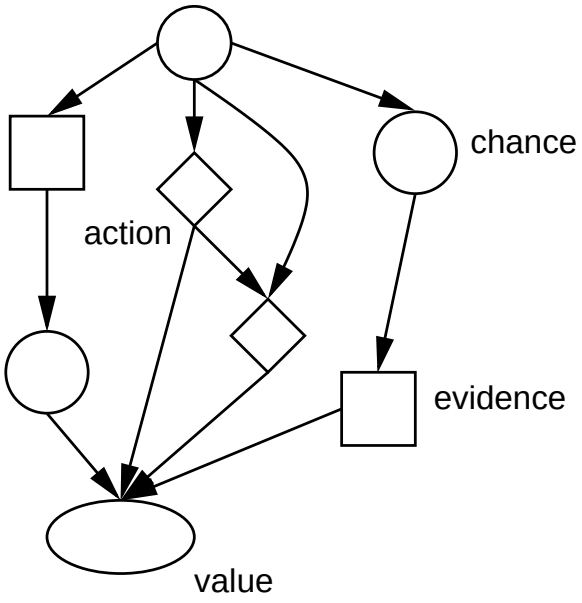

Figure 2.3: An influence diagram. Squares indicate visible variables, and diamonds indicate actions. The unobserved chance nodes are indicated by circles. The value node is an ellipse.

Unlike the localist representation of a POMDP, influence diagrams allow for a compact, factored representation of the dynamics of a process in terms of a set of random variables. The task with an influence diagram is the same as with an MDP or POMDP: To find a policy that optimizes the expected value of the value node. The expectation is taken with respect to the posterior distribution over hidden chance nodes, and with respect to the policy over the action nodes. The posterior can be computed exactly for Gaussian influence diagrams [Shachter and Kenley 1989] but becomes intractable for others as the number of chance and action nodes increases.

There is typically a single value node, and it has no descendants. Since influence diagrams are directed and acyclic, we can work backward from the value node and assign a "time" to each action and chance node, with the value node occurring last in the sequence. By convention, any information available to an action node at time t is also available to all subsequent action nodes; this is called the "no-forgetting" rule.

The no-forgetting rule allows influence diagrams to represent arbitrarily non-Markov processes. Unfortunately this results in policies whose domains increase exponentially with the problem horizon. Although the dynamics of the process can be represented compactly, the policy cannot. Influence diagrams quickly become infeasible for long-running processes.

There are a number of algorithms for solving influence diagrams [Howard and Matheson 1984; Shachter 1986; Cooper 1988; Shachter and Peot 1992]. In the interest of brevity, I will not detail them there. There are two general categories of algorithms: ones that operate directly on the influence diagram [Shachter 1986], and ones that first transform it into an intermediate representation [Howard and Matheson 1984; Cooper 1988; Shachter and Peot 1992]. The former results in a DP-type approach, where the optimal policy is found by working backwards from the value node, eliminating action nodes and integrating out chance nodes. The latter involves performing inference in a series of Bayesian networks [Cooper 1988; Shachter and Peot 1992], or operating on a decision tree [Howard and Matheson 1984].

2.3.2 Dynamic Bayesian Networks

A dynamic Bayesian network is like an influence diagram that obeys the Markov property. A two time-slice dynamic Bayesian network (DBN) represents a system at two time steps [Dean and Kanazawa 1989]. The conditional dependencies between random variables from time t to time $t+1$, and within time step t, are represented by edges in a directed acyclic graph. The conditional probabilities can be stored explicitly, or parameterized by weights on edges in the graph (see figure 2.4).

A DBN allows us to compactly represent a POMDP as a set of variables, and conditional dependencies between variables. If the network is densely-connected then inference in the network is intractable [Cooper 1990]. Approximate inference methods include Markov chain Monte Carlo [Neal 1993], variational methods [Jordan, Ghahramani,

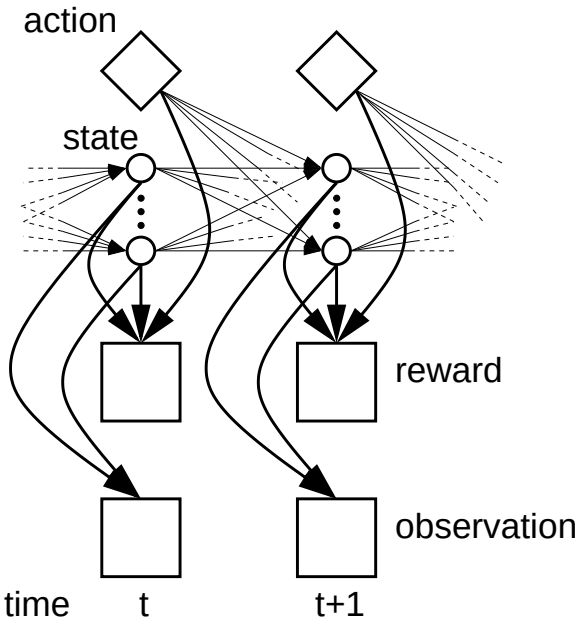

Figure 2.4: A dynamic Bayesian network. Squares indicate visible variables, and diamonds indicate actions. The hidden state variables at time $t+1$ are dependent on the action at time t and on the hidden variable values at time t. The observation depends on the current state variables. The reward depends on the current state and action.

Jaakkola, and Saul 1999], and projective filtering[Boyen and Koller 1998].

We can think of a POMDP as a DBN with just two nodes, representing a random variable at two consecutive time steps that can take on one of a number of values. Thinking in this way, it is clear that in general the problems inherent in representing POMDP value functions carry over to DBNs. But if there is structure in the DBN that allows for compact representations of transition and emission probabilities then this structure can also be used in the representation of the value function. A number of researchers have proposed methods of computing either exact value functions [Boutilier and Poole 1996] or approximate value functions [McAllester and Singh 1999; Sallans 2000] for factored POMDPs , which take advantage of their inherent structure.

In applying a DBN to a large problem there are three distinct issues to disentangle: How well does a parameterized DBN capture the underlying POMDP; how much is the DBN hurt by approximate inference; and how good must the approximation of the value function be to achieve reasonable performance? The characterization of the value

function as piecewise linear and convex in the belief state assumes that the model is known exactly and that the belief state is updated exactly. In other words, the belief state really does characterize the correct probability of being in a particular state of the underlying core Markov process. Interesting questions about approximation arise when these assumptions are not met. Inaccuracies can be introduced if:

1. The model is not known exactly, so belief updates are not exact.

2. Belief updates are too expensive to compute exactly.

3. The value function is too complex to store exactly (e.g., too many linear pieces), so an approximation must be used.

In chapter 3 I will discuss a system where all of the above are true. They will be true in general of any large, complex problem. Others have proposed methods that tackle one or more of these issues. In [McAllester and Singh 1999] an approximate inference method is used in conjunction with a complementary value function approximation. Both approximations use a projection operator, where either the hidden state distribution or the value function from the previous time step are updated exactly, and then projected back into a space of tractable distributions or value functions. They assume that the model is known exactly, and they find bounds on the error of the value function approximation caused by both projection operations. Rodriguez, Parr, and Koller [2000] use a similar projection operator for approximate inference, but with parameterized value function approximations. Because they use reinforcement learning, they do not need to be supplied with the transition and emission probabilities. They use both feedforward MLPs and the smooth value function approximation of Parr and Russell [1995]. Poupart and Boutilier [2000] assume that inference is approximate, and use the resultant error in the value function to select the best approximate inference technique. Again, it is assumed that all transition and emission probabilities are known in advance.

In the following sections I discuss methods of exact and approximate inference, and methods of estimating parameter values in parameterized Bayesian networks and DBNs.

2.3.3 Factored Markov Decision Processes

In subsequent chapters I will focus on factored Markov decision processes (factored MDPs), and factored partially observable MDPs, in which each state and action is represented as a set of variables. Formally, a factored (PO)MDP consists of a set $\{\{\mathcal{S}_\alpha\}_{\alpha=1}^M, \{\mathcal{A}_\beta\}_{\beta=1}^N, \{s_\alpha^0\}_{\alpha=1}^M, P, P_r\}$, where: $\mathcal{S}_\alpha$ is the set (or field) of possible values for state variable α; $\mathcal{A}_\beta$ is the set of possible values for action variable β; s_α^0 is the initial value for state variable α; P is a transition distribution $P(\mathbf{s}^{t+1}|\mathbf{s}^t, \mathbf{a}^t)$; and P_r is a reward distribution $P(r^t|\mathbf{s}^t, \mathbf{a}^t, \mathbf{s}^{t+1})$. A state is an M-tuple and an action is an N-tuple. Additionally a POMDP includes a set of output symbols $\{\mathcal{O}_\kappa\}_{\kappa=1}^K$ and output distributions $P(\mathbf{o}^t|\mathbf{s}^t)$. An output is a K-tuple.

A factored MDP or POMDP can be represented compactly as a (set of) dynamic Bayesian networks . Transition and emission probabilities are parameterized while taking advantage of any conditional independencies between state and action variables. POMDPs expressed in this way can be solved exactly with dynamic programming, while still taking advantage of their compact representation [Boutilier, Dearden, and Goldszmidt 1999]. Instead of computing the value function exactly, value functions can be learned using reinforcement learning techniques while still taking advantage of the structured representation [Boutilier and Dearden 1996; Koller and Parr 1999; St-Aubin, Hoey, and Boutilier 2000].

2.4 Learning and Inference

So far I have focused on the task of learning value functions for fully and partially observable Markov decision processes. For the dynamic programming algorithms, it was

assumed that the learner has access to knowledge about the dynamics of the process. In the following section I discuss learning models of the transition and generation dynamics from data, and making inferences about the state of the core process.

2.4.1 The Expectation-Maximization algorithm

The Expectation-Maximization (E-M) algorithm is an iterative method for learning maximum likelihood parameters of a generative model where some of the random variables are observed, and some are hidden [Dempster, Laird, and Rubin 1977]. The hidden random variables might represent quantities that we think are the underlying causes of the observables. For example, a model designed to explain data consisting of shoe size and reading ability might use age as a hidden variable. The hidden variables could be continuous as in the above example, or discrete as in the case of class labels.

Let $\mathbf{x}$ be the settings of the visible variables, $\mathbf{y}$ be the settings of hidden variables, and let θ be the parameters of the model. As the name implies there are two steps to the algorithm:

Expectation (E) step: Calculate the distribution $P(\mathbf{y}|\mathbf{x};\theta)$ over the hidden variables, given the visible variables and the current value of the parameters.

Maximization (M) step: Compute the values of the parameters θ^* that maximize the expected log-likelihood under the distribution found in the E-step:

$$\begin{aligned} \theta^* &= \operatorname*{argmax}_\theta \left\{ \mathrm{E}\left[\log P(\mathbf{x},\mathbf{y}|\theta)\right]_{P(\mathbf{y}|\mathbf{x})} \right\} \\ &= \operatorname*{argmax}_\theta \left\{ \sum_{\mathbf{y}} P(\mathbf{y}|\mathbf{x}) \log P(\mathbf{x},\mathbf{y}|\theta) \right\} \end{aligned} \tag{2.40}$$

and set $\theta \longleftarrow \theta^*$.

So the E-step involves inferring the distribution over hidden variables, and the M-step involves learning new parameters. It can be shown that in most cases if these two steps

are repeated the true log-likelihood will increase, or stay the same if a local maximum has already been reached.

Notice that the M-step might require solving a difficult non-linear optimization problem. It is sometimes natural to implement a partial M-step instead, where we just find a set of parameters that improve the expected log-likelihood instead of fully maximizing it. For example, gradient ascent-based partial M-steps are quite common. Algorithms that use a partial M-step are called generalized E-M algorithms (GEM), and they are also guaranteed to improve the true likelihood [Dempster, Laird, and Rubin 1977].

The E-step can also be very difficult, depending on the form of the posterior. Sometimes we must resort to approximate inference, where instead of finding the true posterior, we find an approximation to the true posterior. The approximation is then used to compute the expectation required in the M-step. Inference is central to the problem of learning ML parameters with the E-M algorithm. Performing exact or approximate inference is a very important problem that must be solved both to learn the parameters of a generative model, and to use the model once ML parameters have been found.

2.4.2 Exact inference

In some cases there is no need for approximate inference because we can efficiently compute the correct posterior distribution. Examples include the case of linear-Gaussian models; and singly-connected graphical models, where some variant of probability propagation can be used to compute the posterior correctly [Pearl 1988]. Factor analysis is an example of the former, and the hidden Markov model [Baum and Petrie 1966] is an example of the latter. Kalman filters can be both singly-connected and Gaussian [Kalman 1960]. Unfortunately, inference is provably hard for most other directed acyclic graphical models. That is, the problem of inference in general belief networks is NP-hard [Cooper 1990]. Even apparently simpler problems have been shown to be NP-hard. Finding a single most-likely set of hidden states (the *maximum a posteriori* or MAP states), and

the problem of approximate inference within a given accuracy are known to be NP-hard [Shimony 1994; Dagum and Luby 1993].

A new class of undirected graphical models has recently been proposed whose members are more powerful than those listed above, but still allow for exact inference. These are products of experts (PoE) models [Hinton 1999]. The tradeoff with PoE models is that they do not explicitly define a density (or distribution) over data space. Instead, they define an unnormalized energy function. It is therefore difficult to maximize the log-probability of the data under the model as is typically done with other models (such as linear dynamical systems or hidden Markov models). Instead, Hinton suggests a new learning algorithm, called minimizing contrastive divergence, which is a gradient method based on minimizing energy on the data and maximizing energy on one-step reconstructions of the data. See [Hinton 1999] for more details of using products of experts in an unsupervised setting. I will discuss products of experts in more detail in chapter 4.

The advantages of exact inference are obvious: speed and precision. These properties account for the wide adoption of models in which exact inference is possible. Unfortunately exact inference has typically only been used in a very restricted range of architectures, and these do not seem to be expressive enough to capture complicated phenomena. Products of experts models allow for exact inference, and are expressive enough to capture interesting non-linear structure in data, but they are harder to train than singly-connected or Gaussian models.

2.4.3 Variational methods

Consider a parameterized distribution $\widehat{P}$ over hidden variables. Given a metric that measures the difference between this distribution and the true posterior P we can optimize the parameters of $\widehat{P}$ to approximate P. The approximation $\widehat{P}$ is called a variational approximation, and the parameters are variational parameters. There is a suitable distance

metric called the Kullback-Leibler divergence of $\widehat{P}$ from P:

$$\mathrm{KL}(\widehat{P}\|P) = \int \widehat{P}(\mathbf{y}|\mathbf{x}) \log \frac{\widehat{P}(\mathbf{y}|\mathbf{x})}{P(\mathbf{y}|\mathbf{x})} d\mathbf{y} \tag{2.41}$$

where $\mathbf{y}$ is a vector of hidden unit activities, and $\mathbf{x}$ is an observation.

To evaluate Eq.(2.41) directly we would need to evaluate $P(\mathbf{y}|\mathbf{x})$ which is what we are trying to approximate in the first place. We can get around this difficulty by evaluating the *variational free energy* which is a function of the joint probability of the hidden and visible units $P(\mathbf{x}, \mathbf{y})$. The variational free energy is comparable to "variational free energy" from statistical physics. The joint probability is readily available in directed belief networks.

The free energy is actually an upper bound on the negative log-probability of the data. It is the negative log-probability of the data plus the KL divergence between $\widehat{P}$ and P:[4]

$$F = -\log P(\mathbf{x}) + \mathrm{KL}(\widehat{P}\|P) \tag{2.42}$$

Instead of optimizing the parameters of $\widehat{P}$ by minimizing Eq.(2.41), we can minimize Eq.(2.42), which will have the same effect (since $\log P(\mathbf{x})$ is independent of $\widehat{P}$). It turns out that we can also train the model by minimizing Eq.(2.42) with respect to the model parameters.

Variational methods have several advantages. They allow us to calculate an upper and lower bound on the log-probability of the data under the current generative model [Jaakkola 1997] and they are fast. They are also deterministic which can be an advantage in situations where gradients are shallow and sampling noise might hinder learning. Although variational methods yield approximations that have low variance, they are biased.

The down side of variational methods is that it is not always easy to come up with

[4]The reader should note that the free energy is sometimes defined with opposite sign: $F = \log P(\mathbf{x}) - \mathrm{KL}(Q\|P)$. However, I will use the sign which is consistent with the free energy from statistical physics.

a good approximating distribution. Typically a good variational approximation is very architecture-specific. If care is not taken, the approximating distribution can be too simplistic to capture important features of the posterior, or it can be too complicated to be used in subsequent calculations. Finding good approximating distributions is as much art as science.

Two commonly used variational approximations are the *mean field* approximation, and the *maximum a posteriori* (MAP) approximation. Under a mean field approximation we assume that the hidden units are independent. Instead of having a distribution whose representation requires space exponential in the number of hidden units we have one that only needs linear space. There are also mean-field approximations that do not provide a lower bound on the log-likelihood of the data. Instead, the mean-field parameters are just updated so as to be self-consistent. This simplifies the mean-field equations, but provides only an approximation to the likelihood instead of a lower bound. The self-consistency equations can be solved iteratively.

Under a MAP approximation we assume that $\widehat{P}$ is a single spike of infinite density at a point $\mathbf{y}_m$. The entire mass of the distribution is replaced with this single spike. In this case, the KL divergence between $\widehat{P}$ and P is infinite, so we optimize the parameters of $\widehat{P}$ with respect to the expected energy under $\widehat{P}$, given by:

$$\begin{aligned} E_{\widehat{P}}[\text{energy}] &= E_{\widehat{P}}[-\log P(\mathbf{y}, \mathbf{x})] \\ &= -\int \widehat{P}(\mathbf{y}|\mathbf{x}) \log P(\mathbf{y}, \mathbf{x}) d\mathbf{y} \\ &= -\int \log P(\mathbf{y}, \mathbf{x}) \delta(\mathbf{y} - \mathbf{y}_m) d\mathbf{y} \\ &= -\log P(\mathbf{y}_m, \mathbf{x}) \end{aligned} \tag{2.43}$$

This quantity is proportional to the posterior, so during the optimization of $\widehat{P}$ we move the spike to a point of maximum density under the posterior, thus the name *maximum a posteriori*. This can be a reasonable approximation if the true posterior is unimodal and

sharply-peaked. Both of these approximations are common because they are simple, and they seem to work for a large class of problems. However if the posterior is multimodal or broad they may be insufficient.

It might appear that mean-field and MAP approximations are too simple to give good results when learning model parameters. However, there is a mitigating factor that allows simple approximations to do a reasonable job when they are coupled with model learning: Because we update the model parameters by performing gradient descent in F, the model parameters are adjusted so as to reduce $\mathrm{KL}(\widehat{P}\|P)$ as well as to increase $\log P(\mathbf{x})$. The model parameters change so that the true posterior is brought closer to the approximation.

2.4.4 Monte Carlo methods

Monte Carlo approximations can be used to estimate expectations when we cannot evaluate the posterior explicitly, but we can sample from it. Given N samples $\{\mathbf{x}_1, ..., \mathbf{x}_N\}$ from a distribution $P(\mathbf{x})$, we can approximate the expectation of a function $f(\mathbf{x})$ under P as follows:

$$\begin{aligned} E[f(\mathbf{x})] &= \int f(\mathbf{x})P(\mathbf{x})d\mathbf{x} \\ &\approx \frac{1}{N}\sum_{i=1}^{N} f(\mathbf{x}_i) \end{aligned} \tag{2.44}$$

There are a number of different schemes for generating the set of samples which depend on the form of the distribution P (see [Neal 1993] for a survey of Monte Carlo sampling techniques). In general, Monte Carlo methods are not as dependent on architecture as are variational methods. A single Monte Carlo method can be applied to a broad range of architectures. The disadvantage of Monte Carlo methods is speed: It can take a long time to approximate an expectation to a high degree of accuracy. Also, because the approximations involve taking a random sample from P, we get a noisy

estimate of any quantities of interest.

There is a subset of Monte Carlo methods called Markov chain Monte Carlo (MCMC). Markov chain Monte Carlo sampling techniques are appropriate when we can not sample directly from the distribution of interest. Instead we sample from an ergodic Markov chain whose unique equilibrium distribution is the distribution from which we actually want to sample. After the Markov chain has reached equilibrium we can draw samples from the original distribution of interest.

One drawback of MCMC techniques is that we must in general wait for the chain to reach equilibrium. This can take a long time, and it is sometimes difficult to tell when it happens. Some work has been done on learning and inference with brief sampling [Hinton and Ghahramani 1997; **?**]. This involves taking too few samples to allow the Markov chain to reach equilibrium. Like a variational method, there is pressure for the model parameters to take on values so that a sample from the posterior distribution can be taken after only a few iterations of sampling. Also like a variational method, brief sampling yields a biased estimate of the posterior distribution. Unlike a variational method, brief sampling can be more easily applied to a large range of graphical models. However brief sampling, like MCMC methods in general, do not give us access to a bound on the log-likelihood.

Gibbs Sampling

Gibbs sampling is an example of a Markov chain Monte Carlo sampling technique. Consider a distribution P over some random variables $\mathbf{x} = \{x_1, x_2, ..., x_k\}$. With Gibbs sampling, it is sufficient if we can sample from any one of the variables conditioned on the others: $P(x_j|\{x_i\}_{i\neq j})$. Gibbs sampling proceeds as follows:

1. Initialize $x_1, ..., x_k$ by sampling from some initial distribution P_0.

2. For each $j \in \{1, ..., k\}$, sample from $P(x_j|\{x_i\}_{i\neq j})$.

When sampling from subsequent variables use the newly sampled values of preceding variables.

3. Iterate until convergence to the stationary distribution P_s, which is the distribution of interest.

Of course the method can be made more elaborate. We need not sample from single variables, but can instead sample from subsets, or blocks, of variables. All that is required is that we can sample from the distribution of the block $P(\{x_j\}|\{x_i\})$ conditioned on the other variables. In practice the biggest problem is knowing when the Markov chain has converged to the stationary distribution P_s.

2.4.5 Projective Filtering

This approximate inference method for Markov processes has recently received attention in the machine learning community [Boyen and Koller 1998]. It requires a projection operator which transforms an arbitrary probability distribution (or density for real-valued domains) into a distribution from a subset of simpler distributions $\mathbf{P}_a$. The approximation involves the following steps:

1. Begin at time t with a distribution $\widehat{P}_t \in \mathbf{P}_a$.

2. Use Bayes' rule to update the distribution to P_{t+1} (called belief update). This includes applying the dynamics of the Markov process and incorporating newly observed evidence. Notice that P_{t+1} may now fall outside of $\mathbf{P}_a$.

3. Use the projection operator to get $\widehat{P}_{t+1} \in \mathbf{P}_a$.

The subset $\mathbf{P}_a$ should have the following property: Whatever calculations we want to perform should be tractable when using a member of the subset. For example, if the set of distributions in question were all distributions on n binary variables, just storing the distribution would require space $\mathcal{O}(2^n)$. A typical subset would be all distributions on

the binary variables that can be represented in completely factored form, requiring space of only $\mathcal{O}(n)$ (see figure 2.5). The projection operator in this case is marginalization.

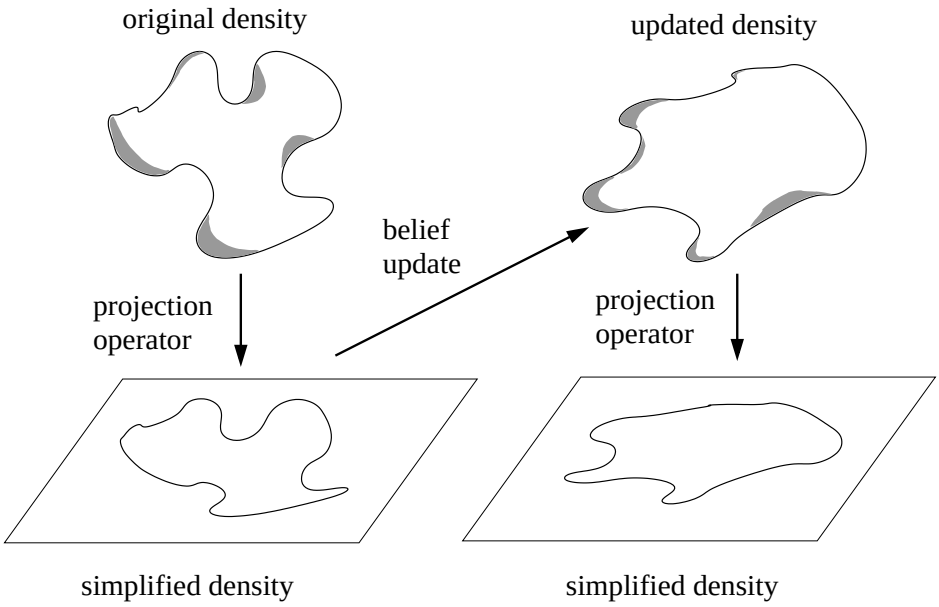

Figure 2.5: Schematic of projection filtering. The original distribution (or density) is projected in to the space of simpler distributions. This simpler distribution is updated according to Bayes' rule, and the result is re-projected into the space of simpler distributions.

Intuitively this method relies on the fact that while an updated distribution might fall outside of the set of simple distributions, it will be "not far away", and can be easily projected back in to the space of simple distributions. Here, "easily" means that the projection operation can be handled with less than worst-case space and time requirements. In the example above a belief update operation should not introduce too many correlations into the fully-factored approximating distribution. It has been shown theoretically that the error in the approximating distribution remains bounded irrespective of the number of projection operations performed [Boyen and Koller 1998] and that planning in a POMDP based on simplified belief states results in a bounded deviation from optimal rewards [McAllester and Singh 1999]. This inference method has also been tested empirically for POMDPs [Rodriguez, Parr, and Koller 2000; Poupart and Boutilier 2001].

Chapter 3

Factored States

In this chapter I present a method for learning compact representations of the core process of a partially observable MDP , and approximating the belief-state action value function. Preliminary results were presented in [Sallans 2000].

3.1 Compact Representations and Approximations

As the decision problems that we want to solve grow larger, we need to use more and more efficient representations for the model of the world. Early research in POMDPs described the state of the world by a single discrete number. This was sufficient for solving gridworld-style problems with a few tens of states or observations. We are now interested in problems involving anywhere from thousands to billions of states and observations. Solution methods for these larger problems should have the following properties:

1. The representation of the core process is compact.

2. There is an efficient way to map from the observation space to internal states (these might be belief states, internal states of finite automata, etc.).

3. We can learning the mapping from experience.

Finally, the solution method should take advantage of the fact that only a tiny subset of possible observations will ever be seen by the system. These will typically lie in some low-dimensional manifold embedded in the high-dimensional observation space (for example, think of visual scenes that actually occur versus all possible pixel settings).

Unsupervised learning with a Bayesian network meets all of these criteria. A Bayesian network can compactly represent the transition and emission probabilities of a factored POMDP The core state is represented as a set of random variables. Inference in the network will allow us to update the belief state given new evidence. Finally, we can use gradient methods to update the parameters of the network based on experience. The learned model will give high probability to the subset of possible observations that are actually seen and ignore the rest.

A two time-slice dynamic Bayesian network (DBN) represents the system at two consecutive time steps [Dean and Kanazawa 1989]. The conditional dependencies between random variables from time t to time $t + 1$, and within time step t, are represented by edges in a directed acyclic graph. The conditional probabilities can be stored explicitly, or parameterized by weights on edges in the graph.

There are a number of challenges to overcome when using a Bayesian network as a model of dynamics. If the transition, emission and reward probabilities are not known a priori, they must be learned by observing interactions between the agent and the world. Similarly, learning the belief-state value function requires observing the statistics of rewards. The two interact because updating the value function depends on what we believe about the dynamics of the process. Finally, if the network is densely-connected then inference is intractable [Cooper 1990]. Approximate inference methods will affect the accuracy of belief updating, which will in turn affect the estimate of the belief state value function and the process parameters. On top of all of this, we will also have to use function approximation for the belief-state value function.

The method presented in this chapter is designed to show that these challenges are not insurmountable. The method consists of learning a compact Bayesian network representation of the system dynamics; learning a belief-state value function based on this representation; and using approximate inference to update the belief state as new evidence is observed. All of these steps are carried out simultaneously as the learning agent interacts with the world.

All of the approximations are coupled, making theoretical analysis difficult. Recent research has investigated subsets of these issues both theoretically and empirically [McAllester and Singh 1999; Poupart and Boutilier 2000; Thrun 2000; Rodriguez, Parr, and Koller 2000; Sallans 2000]. Each of these approaches necessitates making choices about approximate inference, model learning and the value function approximation. I will discuss in detail one such set of choices involving mean-field approximate inference and gradient-based parameter learning.

Ideally a factored representation can be learned quickly, because the model can represent the transition and emission process with few parameters. Further, the factored representation will allow for a core MDP to be parameterized that is simply too large for a localist representation. The factorization assumption can also be carried over to the value function approximation, again eliminating parameters and simplifying the interpretation of hidden variables.

In the first section of this chapter I discuss each of the issues: approximate inference; learning model parameters; and approximating the value function. For each I describe the general problem and detail a specific example: the dynamic sigmoid belief network. In the final sections I discuss simulation results. The final simulation shows the DBN method solving a time-sensitive driving problem with hidden state and an input space of size 2^{180} (although only a small subset of the possible inputs are ever observed). The model used to solve this problem has a core state-space of size 2^{32} represented as 32 binary variables.

3.2 A Dynamic Sigmoid Belief Network

Throughout this chapter I will illustrate the DBN approach with a running example: A fully-connected dynamic extension of the sigmoid belief network [Neal 1992], with K units at each time slice. I will refer to this network as a dynamic sigmoid belief network, or DSBN (see figure 3.1). The random variables s_i are binary, and conditional probabilities

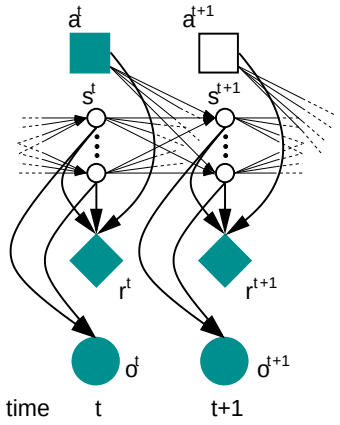

Figure 3.1: Architecture of the dynamic sigmoid belief network. Circles indicate random variables, where a filled circle is observed and an empty circle is unobserved. Squares are action nodes, and diamonds are rewards.

relating variables at adjacent time-steps are encoded in action-specific weights:

$$P(s_i^{t+1} = 1 | \{s_k^t\}_{k=1}^K, a^t) = \sigma\left(\sum_{k=1}^K w_{ik}^{a^t} s_k^t\right) \tag{3.1}$$

where $w_{ik}^{a^t}$ is the weight from the i^{th} unit at time step t to the k^{th} unit at time step $t+1$, assuming action a^t is taken at time t. The nonlinearity is the usual logistic function: $\sigma(x) = 1/(1 + \exp\{-x\})$. Note that a bias can be incorporated into the weights by clamping one of the binary units to 1.

The observed variables are assumed to be discrete; the conditional distribution of an output given the hidden state is multinomial and parameterized by output weights. The probability of observing an output with value l is given by:

$$P(o^t = l | \{s_k^t\}_{k=1}^K) = \frac{\exp\left\{\sum_{k=1}^K u_{lk} s_k^t\right\}}{\sum_{m=1}^{|\mathbf{O}|} \exp\left\{\sum_{k=1}^K u_{mk} s_k^t\right\}} \tag{3.2}$$

where $o^t \in \mathbf{O}$ and u_{lk} denotes the output weight from hidden unit k to output value l.

Given this parameterization, the expected log-likelihood of a new time slice is given by:

$$\begin{aligned}
&\left\langle \log P(\mathbf{s}^{t-1}, \mathbf{s}^t, o^t | \{a\}_1^{t-1}, \{o\}_1^{t-1}) \right\rangle \\
&\quad = \left\langle \mathbf{s}^{t\top} \log \sigma(\mathbf{W}^{a^{t-1}} \mathbf{s}^{t-1}) + (1 - \mathbf{s}^t)^\top \log \left(1 - \sigma(\mathbf{W}^{a^{t-1}} \mathbf{s}^{t-1})\right) \right\rangle_{P(\mathbf{s}^{t-1}, \mathbf{s}^t | \{o\}_1^t, \{a\}_1^{t-1})} \\
&\quad + \left\langle \nu^{t\top} \log \left(\frac{\exp(\mathbf{U}\mathbf{s}^t)}{\sum_i [\exp(\mathbf{U}\mathbf{s}^t)]i} \right) \right\rangle_{P(\mathbf{s}^t | \{o\}_1^t, \{a\}_1^{t-1})}
\end{aligned} \tag{3.3}$$

Here $\mathbf{W}^a$ is the matrix of action-specific weights, $\mathbf{U}$ is a matrix of hidden-to-observation weights, ν^t is a vector of all zeros except for a one in the $o^{t\text{th}}$ place, and $[\cdot]_i$ denotes the i^{th} element of a vector. The expectation is taken over the hidden variables at the current and previous time steps.

This architecture can also be made to accommodate factored observations, by assuming that the probability of each variable in the observation is independent of the others given the hidden variables:

$$\begin{aligned}
&\left\langle \log P(\mathbf{s}^{t-1}, \mathbf{s}^t, \mathbf{o}^t | \{a\}_1^{t-1}, \{\mathbf{o}\}_1^{t-1}) \right\rangle \\
&\quad = \left\langle \mathbf{s}^{t\top} \log \sigma(\mathbf{W}^{a^{t-1}} \mathbf{s}^{t-1}) + (1 - \mathbf{s}^t)^\top \log \left(1 - \sigma(\mathbf{W}^{a^{t-1}} \mathbf{s}^{t-1})\right) \right\rangle_{P(\mathbf{s}^{t-1}, \mathbf{s}^t | \{\mathbf{o}\}_1^t, \{a\}_1^{t-1})} \\
&\quad + \sum_j \left\langle \nu_j^{t\top} \log \left(\frac{\exp(\mathbf{U}_j \mathbf{s}^t)}{\sum_i [\exp(\mathbf{U}_j \mathbf{s}^t)]i} \right) \right\rangle_{P(\mathbf{s}^t | \{\mathbf{o}\}_1^t, \{a\}_1^{t-1})}
\end{aligned} \tag{3.4}$$

Here j indexes visible multinomial variables. We have a seperate emission matrix for each visible variable.

I have chosen this architecture because the discrete output and hidden variables are appropriate to model the tasks that I will consider. Also, the binary hidden variables are relatively easy to interpret. Finally, there is extensive literature on approximate inference schemes for sigmoid belief networks [Neal 1992; Saul, Jaakkola, and Jordan 1996; Jaakkola 1997]. Discrete outputs can easily be replaced by continuous outputs by

allowing each binary hidden unit to gate a real-valued Gaussian output unit [**?**; Sallans 1998].

3.3 Approximate Inference

Inference in a fully-connected Bayesian network is intractable. In chapter 2 I have discussed a number of approximate inference techniques. Each has been used to perform belief update in factored POMDPs [Thrun 2000; Rodriguez, Parr, and Koller 2000; Sallans 2000; Poupart and Boutilier 2000]. I use a mean-field method, with a fully-factored approximating distribution:

$$\phi^t \triangleq P(\mathbf{s}^t|\mathbf{s}^{t-1}, a^{t-1}, o^t) \approx \prod_{k=1}^{K} \mu_k^{s_k^t}(1-\mu_k)^{1-s_k^t} \triangleq \widehat{\phi}^t \tag{3.5}$$

where the μ_k are mean-field parameters.

Given the form of the approximating distribution, we can write down an approximate likelihood for a new time slice, where hidden variable values are replaced by their approximate means:

$$\begin{aligned}\widehat{\mathcal{L}} = {\mu^t}^\top \log \sigma(\mathbf{W}^{a^{t-1}}\mu^{t-1}) + (1-\mu^t)^\top \log\left(1-\sigma(\mathbf{W}^{a^{t-1}}\mu^{t-1})\right) \\ +{\nu^t}^\top \log\left(\frac{\exp(\mathbf{U}\mu^t)}{\sum_i[\exp(\mathbf{U}^{o'}\mu^t)]_i}\right)\end{aligned} \tag{3.6}$$

The mean-field energy is the negative approximate log-likelihood of the approximating distribution: $E = -\widehat{\mathcal{L}}$.

To set the value of mean-field parameter i, we compute the probability of variable s_i being one, given that the rest of the mean field parameters are held fixed. This is just related to the difference in energies when s_i takes on a value of one or zero:

$$P(s_i^t = 1|o^t, \{\mu_k^t\}_{k\neq i}) = \frac{P(o^t, s_i^t = 1|\{\mu_k^t\}_{k\neq i})}{P(o^t, s_i^t = 1|\{\mu_k^t\}_{k\neq i}) + P(o^t, s_i^t = 0|\{\mu_k^t\}_{k\neq i})} \tag{3.7}$$

$$= \frac{\exp\{-E^{s_i=1}\}}{\exp\{-E^{s_i=1}\} + \exp\{-E^{s_i=0}\}} \tag{3.8}$$

$$= \frac{1}{1 + \exp\{-E^{s_i=0} + E^{s_i=1}\}} \tag{3.9}$$

$$= \sigma(E^{s_i=0} - E^{s_i=1}) \tag{3.10}$$

where $E^{s_i=1}$ or $E^{s_i=0}$ denote the energy when s_i takes on a value of one or zero respectively. We then equate the mean-field parameter and this probability:

$$\mu_i^t = \sigma(E^{s_i=0} - E^{s_i=1}) \tag{3.11}$$

This results in K coupled non-linear equations, one for each mean-field parameter. We solve for a self-consistent set of mean-field parameters by iterating these equations.

$$\mu_i^t \leftarrow \sigma(E^{s_i=0} - E^{s_i=1}) \tag{3.12}$$

The parameters converge to self-consistent values within a few iterations. To demonstrate this, I ran the approximate inference algorithm on 10 000 randomly generated DSBNs. The transition and emission parameters were chosen from $\mathcal{N}(0, 1)$. There were 10 binary hidden state variables, and there was one observable variable that could take on one of 10 values. For each random network, the mean-field parameters were initialized randomly from a uniform distribution in the range $[0, 1]$. Eq.(3.12) was then iterated 40 times. The squared difference between parameter values from one iteration to the next was recorded. The median iteration-to-iteration difference across the 10000 trials is shown in figure 3.2. The parameters converge within 15 iterations.

More complex equations can be used which introduce additional parameters and give lower bounds to the true log-likelihood [Jaakkola 1997]. Also, we can assume that the distribution is not fully factored. These "structured" variational approximations can take advantage of the specific connectivity of the graph and result in more efficient and

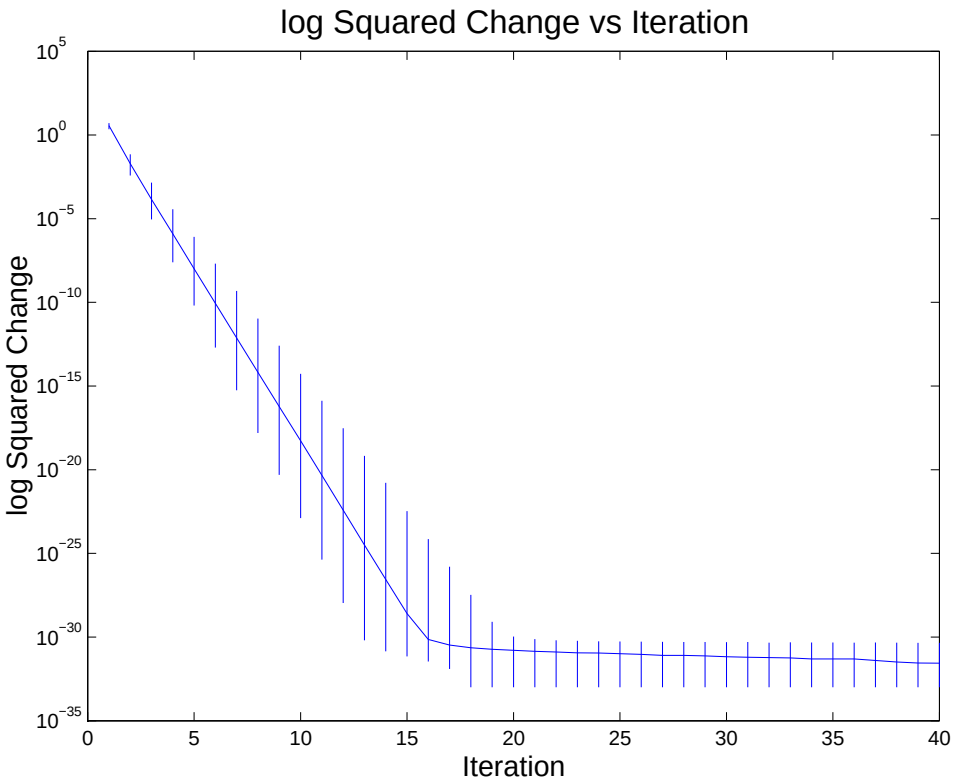

Figure 3.2: Iteration-to-iteration squared difference in mean-field parameters. The graph shows the median differences over 10 000 randomly generated DSBNs. The vertical lines contain 80 % of the data.

accurate inference algorithms [Ghahramani and Jordan 1997].

The values of the mean-field parameters at time $t-1$ are held fixed while computing the values for step t. Because all calculations are done on-line while the learner is interacting with the world, I do not go back and revise belief state estimates based on future evidence. This is analogous to running only the forward portion of the forward-backward algorithm for hidden Markov models. Note, however, that the values of future belief states are still taken in to account when estimating the values of current belief states (see section 3.5).

3.4 Learning Process Parameters

In general, maximum likelihood parameters of a dynamic Bayesian network can be learned using the E-M algorithm. Common examples includes system identification for linear dynamical systems [Shumway and Stoffer 1982], and the Baum-Welsh algorithm for hidden Markov models [Baum, Petrie, Soules, and Weiss 1970]. These methods involve seeing an entire data sequence, computing the posterior distribution over hidden variables at all times given the sequence, and then updating the parameters according to the statistics of the posterior distribution.

We want to identify the parameters of an infinite-horizon POMDP where a completed data sequence is not available. We could use a windowed approach, and apply full E-M updates to the entire window. Others have suggested on-line E-M algorithms that keep a running average and make updates to the parameters as in the batch E-M case [Sato and Ishii 1999]. Instead I use an online gradient-based method. The parameters of the model are optimized online using stochastic gradient ascent in the approximate expected log-likelihood:

$$\Delta\theta \;\propto\; \frac{\partial}{\partial\theta}\widehat{\mathcal{L}} \tag{3.13}$$

where θ are the model parameters.

For the DSBN the transition parameter update is found by differentiating the (single time-slice) approximate likelihood:

$$\begin{aligned}\frac{\partial\widehat{\mathcal{L}}}{\partial\mathbf{W}^{a^{t-1}}} &= \frac{\partial}{\partial\mathbf{W}^{a^{t-1}}}\Big[{\mu^t}^\top \log\sigma(\mathbf{W}^{a^{t-1}}\mu^{t-1}) + (1-\mu^t)^\top \log\Big(1-\sigma(\mathbf{W}^{a^{t-1}}\mu^{t-1})\Big) \\ &\qquad + {\nu^t}^\top \log\left(\frac{\exp(\mathbf{U}\mu^t)}{\sum_i[\exp(\mathbf{U}\mu^t)]_i}\right)\Big] \qquad (3.14)\end{aligned}$$

$$= \mu^t\cdot\Big(1-\sigma(\mathbf{W}^{a^{t-1}}\mu^{t-1})\Big)\,{\mu^{t-1}}^\top - (1-\mu^t)\sigma(\mathbf{W}^{a^{t-1}}\mu^{t-1}){\mu^{t-1}}^\top \qquad (3.15)$$

$$
\begin{aligned}
&= \mu^t {\mu^{t-1}}^\top - \mu^t \cdot \sigma(\mathbf{W}^{a^{t-1}}\mu^{t-1}){\mu^{t-1}}^\top \\
&\qquad -\sigma(\mathbf{W}^{a^{t-1}}\mu^{t-1}){\mu^{t-1}}^\top + \mu^t \cdot \sigma(\mathbf{W}^{a^{t-1}}\mu^{t-1}){\mu^{t-1}}^\top \qquad (3.16)\\
&= \mu^t {\mu^{t-1}}^\top - \sigma(\mathbf{W}^{a^{t-1}}\mu^{t-1}){\mu^{t-1}}^\top \qquad (3.17)\\
&= \left(\mu^t - \sigma(\mathbf{W}^{a^{t-1}}\mu^{t-1})\right){\mu^{t-1}}^\top \qquad (3.18)
\end{aligned}
$$

The emission parameter update is found similarly:

$$
\begin{aligned}
\frac{\partial \widehat{\mathcal{L}}}{\partial \mathbf{U}} &= \frac{\partial}{\partial \mathbf{U}}\Big[{\mu^t}^\top \log\sigma(\mathbf{W}^{a^{t-1}}\mu^{t-1}) + (1-\mu^t)^\top \log\left(1-\sigma(\mathbf{W}^{a^{t-1}}\mu^{t-1})\right) \\
&\qquad +{\nu^t}^\top \log\left(\exp(\mathbf{U}\mu^t)\right) - \log\left(\sum_i [\exp(\mathbf{U}\mu^t)]_i\right)\Big] \qquad (3.19)\\
&= \nu^t {\mu^t}^\top - \frac{\partial}{\partial \mathbf{U}} \log\left(\sum_i [\exp(\mathbf{U}\mu^t)]_i\right) \qquad (3.20)\\
&= \nu^t {\mu^t}^\top - \frac{\exp(\mathbf{U}\mu^t)}{\sum_i [\exp(\mathbf{U}\mu^t)]_i}{\mu^t}^\top \qquad (3.21)\\
&= \left(\nu^t - \frac{\exp(\mathbf{U}\mu^t)}{\sum_i [\exp(\mathbf{U}\mu^t)]_i}\right){\mu^t}^\top \qquad (3.22)
\end{aligned}
$$

The transition parameter update is therefore:

$$
\mathbf{W}^{a^{t-1}} \leftarrow \mathbf{W}^{a^{t-1}} + \alpha_W \left[\mu^t - \sigma\left(\mathbf{W}^{a^{t-1}}\mu^{t-1}\right)\right] \cdot {\mu^{t-1}}^\top \qquad (3.23)
$$

and the emission parameter update is:

$$
\mathbf{U} \leftarrow \mathbf{U} + \alpha_U \left[\nu^t - \frac{\exp\{\mathbf{U}\mu^t\}}{\sum_i [\exp\{\mathbf{U}\mu^t\}]_k}\right] \cdot {\mu^t}^\top \qquad (3.24)
$$

where α_W and α_U are learning rates.

The gradient approach has the advantage that if the process is not stationary, but is slowly varying, the parameters of the model can adapt to track the changing dynamics. For quickly changing dynamics, a switching model might be more appropriate [Ghahramani and Hinton 1998]. Also, the parameter update rules are simple and easy

to compute. Of course the disadvantage is that only small steps can be taken to update the model parameters, and a learning rate must be chosen a priori.

3.4.1 Automatic Relevance Determination

One difficulty in applying latent-variable models is to decide on the dimensionality of the latent space. It would be useful for the model to adapt its structure to the process being modeled. There are two ways that the model structure can be altered: Additional hidden units can be added to accommodate complexity; or hidden units can be pruned if they are not required. Both methods have been used to learn dynamical models of time series data [Stolcke and Omohundro 1993; **?**] and POMDPs [Chrisman 1992; McCallum 1995]. I will use the latter approach in the form of automatic relevance determination (ARD)[MacKay 1994; Neal 1996].

Informally, the hope is to discover which hidden variables are relevant in explaining the dynamics or emissions of the process. The variables that are not relevant will be pruned away. This technique can been implemented as a form of Bayesian structure learning where a prior Gaussian distribution is placed on the weights favouring small magnitudes. The key point with ARD is that each input unit has its own prior variance parameter. Small variance suggests that all weights leaving the unit will be small, so the unit will have little influence on subsequent values. Large variance indicates that the unit is important in explaining the data.

I do not implement full ARD by approximating the posterior over model parameters. Instead, I use an update scheme inspired by ARD [Neal 1998]. Given the derivative of the approximate likelihood $\frac{\partial \widehat{\mathcal{L}}}{\partial w_{ij}^a}$ the modified update becomes:

$$w_{ij}^a \leftarrow w_{ij}^a + \alpha_W \frac{r_i}{2} \frac{\partial \widehat{\mathcal{L}}}{\partial w_{ij}^a} - \alpha_W \delta_W w_{ij}^a \tag{3.25}$$

Here α_W is a learning rate, δ_W parameterizes the strength of the decay toward zero,[1] and r_i is given by:

$$r_i = \frac{G_i^4}{\max_k G_k^4} \qquad\qquad G_i^2 = \sum_j \left(\frac{\partial \widehat{\mathcal{L}}}{\partial w_{ij}^a} \right)^2 \tag{3.26}$$

Intuitively if a weight leading from a hidden unit is important in modeling the process, then early in learning the gradient of $\widehat{\mathcal{L}}$ with respect to this weight will be large. All of the weights leading from this unit will use a higher learning rate. Conversely, if no weights leading from a unit are important then none of the weights will get a high learning rate. In this case the last term in Eq.(3.25) will move the weights toward zero. After the role of a hidden unit is learned, the gradient will decrease. At some point an equilibrium will be reached between the size of the gradient and the strength of the weight decay.

Typically the goal of regularization is to prevent overfitting. Since we essentially have unlimited data for training (subject to training time limitations) this is not really an issue here. It is more important in this context to avoid bad local minima of the likelihood. As we will see in the experimental results section, the quality of the learned model can vary widely. By allowing the model to start with a large state space and pruning with ARD we can avoid local minima and get more robust and consistent results. In practice training DSBNs with ARD resulted in better performance for large problems.

3.5 Approximating the Value Function

Computing the optimal value function is also intractable. If a factored state-space representation is appropriate, it is natural (if a bit extreme) to assume that the action value function can be decomposed in the same way:

$$Q(P_{\mathbf{s}^t}, a^t) \quad \approx \quad \sum_{k=1}^{K} Q_k(\mu_k^t, a^t) \quad \stackrel{\triangle}{=} \quad Q_F(\mu^t, a^t) \tag{3.27}$$

[1] In [Neal 1998] this method was used in conjunction with early stopping to keep the magnitude of irrelevant weights small. Early stopping is not appropriate for infinite-horizon POMDPs, so I use global weight decay instead.

The main reason for making such an assumption is because it is a simple approximation and is easily implemented. It may not perform well in practice. In fact, in the simulation results that follow (see section 3.6), I find that the best performing value function approximator is the one that does not make this factorization assumption.

This simplifying assumption is still not enough to make finding the optimal value function tractable. Even if the states were completely independent, each Q_k would still be piecewise-linear and convex, with the number of pieces scaling exponentially with the horizon. I test several approximate value functions. Each function approximator and its parameter update equations are given below.

3.5.1 Polynomial approximators

The first and simplest approximator is linear:

$$Q_F(\mu^t, a^t) \;=\; \sum_{k=1}^{K} q_{k,a^t}\ \mu_k^t \;=\; [\mathbf{Q}]_{a^t}{}^{\top} \cdot \mu^t \tag{3.28}$$

I also use a similar quadratic approximation:

$$\begin{aligned} Q_F(\mu^t, a^t) &= \sum_{k=1}^{K} \left(m_{k,a^t}\ (\mu_k^t)^2 + q_{k,a^t}\ \mu_k^t \right) + b_{a^t} \\ &= [\mathbf{M}]_{a^t}{}^{\top} \cdot \left(\mu^t \odot \mu^t \right) + [\mathbf{Q}]_{a^t}{}^{\top} \cdot \mu^t + [\mathbf{b}]_{a^t} \end{aligned} \tag{3.29}$$

Where $\mathbf{M}$, $\mathbf{Q}$ and $\mathbf{b}$ are parameters of the approximations. The symbol $\odot$ denotes element-wise vector multiplication.

In all cases I update each term of the approximation with a modified Q-learning rule [Watkins and Dayan 1992], which corresponds to a delta-rule where the target for input μ^t is $r^t + \gamma \max_a Q_F(\mu^{t+1}, a)$. For the linear approximator the update is found by

differentiating the value function:

$$\begin{aligned}
\Delta q_{k,a^t} &\propto \left[r^t + \gamma \max_a Q_F(\mu^{t+1}, a) - Q_F(\mu^t, a^t)\right] \frac{\partial Q_F(\mu^t, a^t)}{\partial q_{k,a^t}} && (3.30)\\
&= \left[r^t + \gamma \max_a Q_F(\mu^{t+1}, a) - Q_F(\mu^t, a^t)\right] \mu_k^t && (3.31)\\
&= \mu_k^t E_B^Q && (3.32)
\end{aligned}$$

where E_B^Q is the Q-learning TD error:

$$E_B^Q = r^t + \gamma \max_a Q_F(\mu^{t+1}, a) - Q_F(\mu^t, a^t) \tag{3.33}$$

The quadratic update rule is found similarly:

$$\begin{aligned}
m_{k,a^t} &\leftarrow m_{k,a^t} + \alpha\ (\mu_k^t)^2\ E_B^Q\\
q_{k,a^t} &\leftarrow q_{k,a^t} + \alpha\ \mu_k^t\ E_B^Q\\
b_{a^t} &\leftarrow b_{a^t} + \alpha\ E_B^Q
\end{aligned} \tag{3.34}$$

Here α is a learning rate and γ is the temporal discount factor.

3.5.2 Smooth Partially Observable Value Approximation

I also tried a factored version of the smooth partially observable value approximation (SPOVA) of Parr and Russell [1995]:

$$\begin{aligned}
Q_F(\mu^t, a^t) &= \sum_{k=1}^{K} \left(\sum_{j=1}^{J} (q_{k,a^t}^j \mu_k^t)^{k_m} \right)^{\frac{1}{k_m}}\\
&\triangleq \sum_{k=1}^{K} Q_k^S(\mu^t, a^t)
\end{aligned} \tag{3.35}$$

Here multiple linear segments (indexed by j) are used to approximate the value function. The linear segments are combined with a differentiable softmax operation. The line

segments $\mathbf{q}^j$ are optimized during learning. The parameter k_m can be adjusted over the course of learning. As the parameter k_m is increased, the approximation becomes closer to a max operation.

Notice that the sum inside the root must be positive. Parr and Russell suggest adding an additional parameter inside the softmax to force all values to be positive:

$$Q_F(\mu^t, a^t) = \sum_{k=1}^{K} \left(\sum_{j=1}^{J} (q^j_{k,a^t} \mu^t_k + \beta^j)^{k_m} \right)^{\frac{1}{k_m}} \tag{3.36}$$

Instead I constrain all of the q^j_{k,a^t} to be positive during optimization, and add an action-specific offset b_{a^t}:

$$Q_F(\mu^t, a^t) = \left(\sum_{k=1}^{K} \left(\sum_{j=1}^{J} (q^j_{k,a^t} \mu^t_k)^{k_m} \right)^{\frac{1}{k_m}} \right) + b_{a^t} \tag{3.37}$$

The SPOVA update rule is derived in [Parr and Russell 1995]:

$$q^j_{k,a} \leftarrow q^j_{k,a} + \alpha \frac{E^Q_B \mu^t_k (\mathbf{q}^{j\top}_a \mu^t)^{k_m - 1}}{Q^S_k(\mu^t, a)^{k_m - 1}} \tag{3.38}$$

The update rule for the global offset is the same as in the linear case.

3.5.3 Multilayer Perceptron

Finally, I also used a feedforward multilayer perceptron without the factorization assumption.[2] The MLP approximator is learned using backpropagation [Rumelhart, Hinton, and Williams 1986]. The input to the network is the (estimated) belief state, and the output is a vector of values for all possible actions. I use a feedforward network with one hidden layer. The hidden units have sigmoid activation functions, and the output units are

[2] In other words the function is not decomposed into K terms, one for each hidden random variable. The function approximator still takes the fully-factored approximate belief state as its input.

linear:

$$\begin{aligned} H_P^j(\mu^t) &= \sigma(\sum_{k=1}^{K} v_{kj}\mu_k^t) \\ Q_P(\mu^t, a^t) &= \sum_{j=1}^{J} u_{ja^t} H_P^j(\mu^t) \end{aligned} \tag{3.39}$$

Here v_{kj} and u_{ja^t} denote the input-hidden and hidden-output weights respectively. The activity of the j^{th} hidden unit is denoted by H_P^j. The hidden unit activity has been written in this way to emphasize that it is a function of the belief state μ. The update rule is again simply the delta rule with the TD error used as the error function:

$$\begin{aligned} u_{ja^t} &\leftarrow u_{j,a^t} + \alpha\ H_P^j\ E_B^Q \\ v_{kj} &\leftarrow v_{kj} + \alpha\ \mu_k^t\ H_P^j\ (1 - H_P^j) E_B^Q \end{aligned} \tag{3.40}$$

See [Bertsekas and Tsitsiklis 1996] for examples of using MLPs for reinforcement learning with real-valued states.

3.5.4 Eligibility Traces

So far I have discussed temporal difference algorithms which use only the current reward signal to estimate the current TD error. There are other algorithms that use the reward signal over several time steps to estimate error at the current time. For example, full Monte Carlo algorithms estimate the value of the current state by summing all of the rewards from this state to the end of the task.

We can consider an intermediate algorithm. An "n-step backup" [Sutton and Barto 1998] estimates value by summing the reward signal for n steps followed by the value function:

$$E_B^n = r^t + \gamma r^{t+1} + \gamma^2 r^{t+2} + ... + \gamma^{n-1} r^{t+n-1} + \gamma^n V(s^{t+n}) - V(s^t)$$

$$= R_n^t - V(s^t) \tag{3.41}$$

By averaging n-step backups for different values of n, we can get different TD error signals. The family of TD errors parameterized by λ average these errors in a particular way:

$$E_B^{T(\lambda)} = (1-\lambda)\sum_{n=1}^{\infty} \lambda^{n-1} R_n^t - V(s^t) \tag{3.42}$$

At $\lambda = 0$, the learner gets a TD error signal by combining the immediate reward with its estimate of future value. This is the TD error discussed previously:

$$\begin{aligned} E_B^1 &= r^t + \gamma V(s^{t+1}) - V(s^t) \\ &= R_1^t - V(s^t) \end{aligned} \tag{3.43}$$

Intuitively, λ controls the bias-variance trade-off in learning the value function. As λ moves from 0 to 1 the estimate of the value function moves from one based on its own (biased) value estimate at future states toward one based on (unbiased) Monte Carlo estimates of total return. Eligibility traces are a way of implementing TD(λ) reinforcement learning methods for $\lambda \in (0, 1)$. The name comes from the fact that eligibility traces keep track of what parameters are eligible to be updated and which are not.

Eligibility traces for SARSA are straight forward. Because the same policy is used at all times, and this is the policy for which the action value function is being learned, we can keep a decaying count of the number of times each state is visited. Two types of traces have been suggested: Accumulating traces and replacing traces. I will only consider accumulating traces here. Following the notation of [Sutton and Barto 1998],

the accumulating trace is given by:

$$e^t(s,a) = \begin{cases} \gamma\lambda e^{t-1}(s,a) + 1 & \text{if } s = s^t \text{ and } a = a^t; \\ \gamma\lambda e^{t-1}(s,a) & \text{otherwise.} \end{cases} \tag{3.44}$$

for all s, a. If a state-action pair is revisited before the trace has decayed to zero then a trace of greater than one will result.

The update (in the table-lookup case) is:

$$Q(s,a) \leftarrow Q(s,a) + \alpha e^t(s,a) E_B^S \tag{3.45}$$

for all s, a. Here E_B^S is the TD (SARSA) error:

$$E_B^S = r^t + \gamma Q(s^{t+1}, a^{t+1}) - Q(s^t, a^t) \tag{3.46}$$

There are issues in applying eligibility traces to Q-learning. Eligibility traces store the frequency with which state-action pairs are visited under the policy being executed. Q-learning is off-policy. It directly learns the values of the greedy policy, regardless of what policy is being followed. Unfortunately most Q-learning algorithms will not follow the greedy policy with respect to the current action value function during learning. They will sometimes execute exploratory actions instead. The eligibility trace will be valid only until such a non-greedy action is executed.

In the implementation of eligibility traces proposed by Watkins [1989] the traces are maintained until a non-greedy action is executed. At this point all traces are reset to zero. Finally the trace for the current state-action pair is incremented by one. Let $\delta_s^t = 1$ if $s = s^t$ and $\delta_a^t = 1$ if $a = a^t$. Then:

$$e^t(s,a) = \delta_s^t \times \delta_a^t + \begin{cases} \gamma\lambda e^{t-1}(s,a) & \text{if } a^t = \text{argmax}_{a'} Q(s^t, a'); \\ 0 & \text{otherwise.} \end{cases} \tag{3.47}$$

The problem here is that early during learning most actions will be exploratory, so the traces will be reset frequently. But it is precisely during early learning, when the action value function is inaccurate, that the traces will be most useful. Another implementation is suggested by [Peng and Williams 1994]. Although more complex, it has the advantage that the traces are maintained regardless of what action is executed. It also appears to work well in practice. Unfortunately, their implementation is only approximate. It does not have the convergence guarantee that Watkins' implementation enjoys. Finally, a "naive" method is suggested by Sutton and Barto, where the traces are maintained as with Watkins' method, and simply not reset when an exploratory action is taken.

If we treat a POMDP as a belief-state MDP, then a correct way to implement an eligibility trace is to store the trace value for each belief state separately. The value function could then be updated for all belief states simultaneously based on the trace.

Because the belief states lie on a simplex, there will be too many belief states to take this approach. Instead we can maintain a trace for every parameter in the (exact or approximate) value function representation. In general, if a function approximator $Q(\phi, a; \theta)$ is used to represent (or approximate) the belief-state action value function (parameterized by θ) then an eligibility trace will be maintained for each parameter. The update for the accumulating SARSA trace is given by:

$$e^t(\theta, a) = \frac{\partial Q(\phi, a; \theta)}{\partial \theta} \times \delta_a^t + \gamma \lambda e^{t-1}(\theta, a) \tag{3.48}$$

The other updates are similar.

For example, if we use a function approximator which is linear in the belief state, then the derivative with respect to a parameter is given by:

$$\begin{aligned} \frac{\partial Q(\phi, a; \theta)}{\partial \theta} &= \frac{\partial \theta_a^\top \phi}{\partial \theta_{ai}} \\ &= \phi_i \end{aligned} \tag{3.49}$$

and Watkins' Q-learning trace update is [He and Shayman 1999]:

$$e^t(\theta_{ai}, a) = \phi_i^t \times \delta_a^t + \begin{cases} \gamma\lambda e^{t-1}(\theta_{ai}, a) & \text{if } a^t = \text{argmax}_{a'} Q(\phi^t, a'; \theta) \\ 0 & \text{otherwise} \end{cases} \quad (3.50)$$

where i indexes possible values of the hidden state variable.

For a factored POMDP with a linear approximator and mean-field inference, the derivative is:

$$\begin{aligned} \frac{\partial Q(\mu, a; \theta)}{\partial \theta} &= \frac{\partial \theta_a^\top \mu}{\partial \theta_{ai}} \\ &= \mu_i \end{aligned} \quad (3.51)$$

and Watkins' Q-learning trace update is given by:

$$e_i^t(\theta_{ai}, a) = \mu_i^t \times \delta_a^t + \begin{cases} \gamma\lambda e_i^{t-1}(\theta_{ai}, a) & \text{if } a^t = \text{argmax}_{a'} Q(\mu^t, a'; \theta) \\ 0 & \text{otherwise} \end{cases} \quad (3.52)$$

Where i indexes hidden state variables, and μ_i denotes the marginal (approximate) probability of each variable.

I experiment with Sutton and Barto's "naive" accumulating Q-learning trace, and Watkins' accumulating Q-learning trace.

3.6 Simulation Results

3.6.1 New York Driving

The "New York Driving" task involves navigating through slower and faster one-way traffic on a multi-lane highway [McCallum 1995]. The speed of the agent is fixed. Relative to the agent, slower cars approach from the front and faster cars approach from behind.

The agent can change lanes, but "obstacle" cars can not. The goal is to make progress by passing slower cars while staying out of the way of faster cars (see figure 3.3).

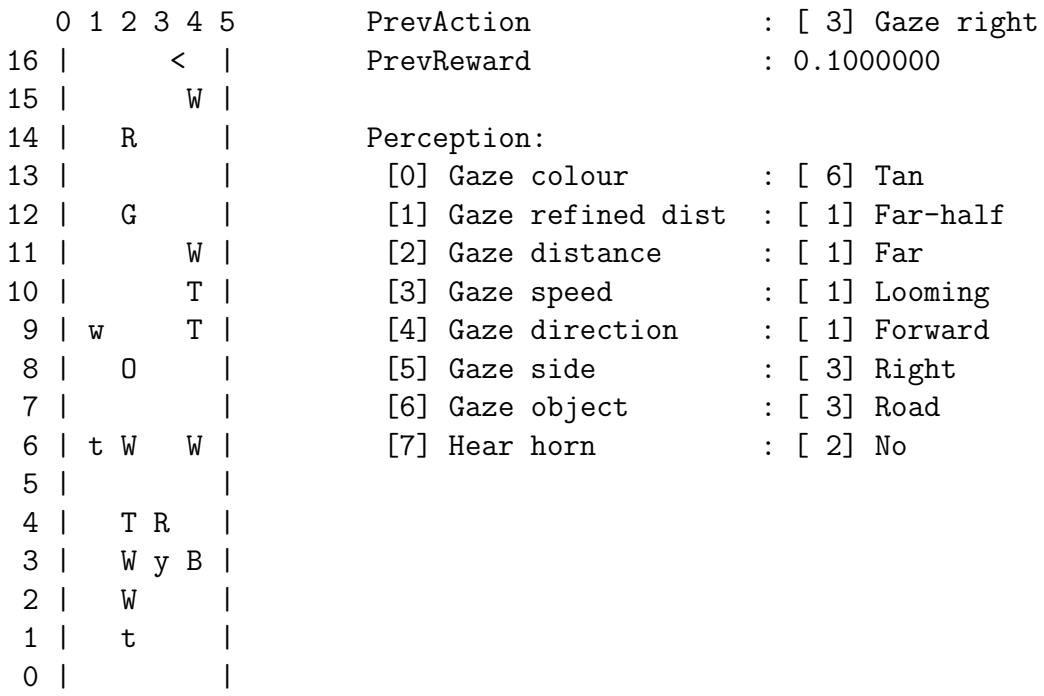

Figure 3.3: "New York Driving" simulator.

The road is divided into four lanes. The agent can see 28 meters in front and behind. Distances are discretized into four meter road sections: eight in front of the agent, eight behind the agent, and one occupied by the agent. Slow cars move at 12 meters/second; fast cars move at 20 meters/second; and the agent moves at 16 meters/second. Time is discretized into 0.5 second intervals. Every two time steps slow cars move back one section and fast cars move forward one section relative to the agent's car.

When a fast car enters the road section behind a slower vehicle, it slows down to match the speed of the blocking car, and begins honking its horn. So fast cars that end up behind slow obstacle cars in effect become slow obstacle cars. Fast cars that are behind the agent's car will slow to 16 meters/second and honk their horn until the agent changes lanes. At this point they will speed up to 20 meters/second and continue on

their way.

The agent has access to four information-gathering actions: "look-left", "look-center", "look-right" and "look-back". It also has one proactive action: "change-lanes" to the current "gaze lane". Looking backward looks backward using the current gaze size: left, center or right. This "current gaze side" is part of the visible state. So going from looking forward-left to backward-right is a two-step operation: The agent must first look right and then look back (or back then right). Changing lanes also requires two actions: First the agent looks into a lane, and then executes "change to gaze lane". Changing lanes while looking "forward-center" has no effect. Lane changes work during both forward and backward gazing. A lane change ends with the agent looking forward into its current lane. During lane changes the agent still makes its normal forward progress at 16 meters/second.

When looking into a lane, the agent's gaze follows that lane from the agent forward or backward until it hits either an obstacle car or the horizon. If there are multiple obstacle cars in a lane, the agent will only see the one closest to it in the current gaze lane.

At each time step the agent has access to some visible world state. The visible state consists of eight discrete variables. The variables and their possible values are summarized in Table 3.1 (from [McCallum 1995]).

Table 3.1: Sensory input for the New York driving task.

Dimension	Size	Values
Hear horn	2	yes, no
Gaze object	3	car, shoulder, road
Gaze side	3	right, center, left
Gaze direction	2	front, back
Gaze speed	2	looming, receding
Gaze distance	3	far, near, nose
Gaze refined distance	2	far-half, near-half
Gaze colour	6	red, blue, yellow, white, gray, tan

Hear horn This is "yes" if a fast car is honking its horn behind the agent, and "no" otherwise.

Gaze object This reports to the agent the type of object currently in view. If the agent is looking off the side of the highway, it sees "shoulder". If it looks at a car it sees "car". Otherwise it sees "road".

Gaze side, Gaze direction The agent has a restricted field of view, and can only see what is in its current lane, or in one lane on either side. The side at which it is currently looking is reported in the "gaze side" variable. It can also look either forward or backward in the chosen lane. This is reported in "gaze direction".

Gaze speed The agent can tell the relative speed of an object by the "gaze speed". Road, shoulder and cars moving toward the agent "loom". Road, shoulder and cars moving away from the agent "recede".

Gaze distance, Gaze refined distance The distance to the object is reported in these two variables. The 8 road sections in front of or behind the agent are broken into three groups: Nose (from 0-8 meters); Near (from 8-12 meters); and Far (beyond 12 meters). The "refined distance" variable further breaks each these discretized distances into two sections. The refined distance can take on the values "far-half" and "near-half'. For example if the state is "Near", "far-half", then a car is somewhere from 10-12 meters from the agent. The intent is that the agent can get either a rough or fine idea of distance by paying attention to just the "gaze distance" or to both "distance" and "refined distance". "Road" and "shoulder" objects always appear at the horizon (i.e.: gaze-distance "Far" and refined-distance "Far half").

Gaze colour Each object is assigned a colour at random. The Road and shoulder can be one of white, gray, tan or yellow. Cars can be any of the six colours. Note that

the colour is not very informative. Mostly it serves to increase the size of the state space and add noise to observations.

When the agent approaches a slower car from behind, it does not slow down or collide with the slow car. Instead it manages to scrape past the slow car in the same lane. McCallum calls this a "NYC squeeze".

A reward is delivered to the agent at each time step. If the agent is "squeezing" past a slow car, it receives a reward of -10. If it is being honked at by a faster car, it receives a reward of -1. Otherwise it receives a positive reward of $+0.1$. The goal of the agent is to change lanes to pass slower cars and stay out of the way of faster cars.

New cars appear randomly at each time step. Fast cars appear from behind the agent, and slow cars appear in front of the agent. At each time step there is a probability of 0.5 that a new car will appear. Fast and slow cars have an equal probability of appearing. Cars are distributed into lanes according to their speed: From right to left, slow cars appear in each of the four lanes with probability 0.4, 0.3, 0.2 and 0.1. Fast cars follow the same distribution, but from left-to-right.

The simulator was equipped with a simple character-based output modeled on the one shown in [McCallum 1995]. Figure 3.3 shows the simulator in action. Capital letters represent slow vehicles and small letters represent fast vehicles. The letter indicates vehicle colour. The letter "O" indicates the agent. The symbol "<" points to the current gaze position (row 16 column 3 (forward-right) in the example).

The New York Driving task was used to test several combinations of inference techniques and value function approximators. A table-based Q-learner and a human driver were also trained on the visible state for comparison purposes. The human driver (the author) was trained on a character-based simulator similar to figure 3.3.[3] The different algorithms are summarized in Table 3.2, and described in the following paragraphs.

[3]Of course, the summary of state variables on the right was omitted. Only the "hear horn" value was shown. Also, the only car shown was the one in the current gaze position (if any). All other state information was removed from the display.

The performance of a number of algorithms and approximations was measured on the task: a random policy; Q-learning on the sensory inputs; a model with a localist representation (i.e. the hidden state consisted of a single multinomial random variable) with various approximate value functions; the DSBN with mean-field inference and approximate value functions; and a human driver. The localist representation used non-factored versions of the value function approximators. The localist method and the DSBN used a factored multinomial evidence representation like the one described in Eq.(3.4). The non-human algorithms were trained 20 times each. Each training run lasted for 100 000 iterations. The human driver (the author) was trained for 1000 iterations using the character-based graphical display, with each iteration lasting 0.5 seconds.

The results for McCallum's U-Tree algorithm are included for comparison [McCallum 1997]. The U-Tree algorithm combines a linear value-function approximator with a decision-tree-based representation of the core process state. The decision tree is learned from experience and makes distinctions based on subsets of the observed variables at various points in the past. The value function is learned by dynamic programming on a model of the dynamics based on the learned state representation.

Also, Glickman and Sycara [2001] have recently reported results on the New York driving task. They use a direct policy method, where the policy is represented as recurrent neural network. Holding the network structure fixed, they optimize the parameters of the network with a genetic algorithm. They report results for two network architectures. The first is a network with stochastic binary units. The second is a network with deterministic sigmoid units. [4]

Stochastic policies were used for all RL algorithms, with actions being chosen from a

[4]These results are reported as follows [Glickman and Sycara 2001]: For the stochastic network, performance was $-245.3 + 179.9$ on 9 out of 10 runs, and $-4193.1 + 450.4$ one 1 out of 10 runs. For the sigmoid network performance was $-172.1 + 225.5$ on 4 out of 10 runs, and $-4193.1 + 450.4$ on 6 out of 10 runs. The intervals are 95% confidence intervals. The results shown in figure 3.4 are samples of 20 scores generated from mixtures of two Gaussians with the appropriate means, variances and mixing proportions

Boltzmann distribution with temperature decreasing over time:

$$P(a^t|\mu^t) = \frac{1}{Z_B} \exp\{Q_F(\mu^t, a^t)/T\} \tag{3.53}$$

In all cases the learning rate was reduced from 0.1 to 0.01 and the exploration temperature was reduced from 1.0 to 0.1 exponentially over the course of the task.

The DSBN had 16 hidden units per time slice, and the localist model used a multinomial with 16 states. In other words both models were matched for number of parameters. The Q-learner had a table representation with 12960 entries (2592 states $\times$ 5 actions). After training, the performance on the final 5000 time steps was examined for each algorithm. The human driver was tested for 2000 time steps, and the results were renormalized for comparison with the other methods.[5] The results are shown in figure 3.4. Note that all results were negative, with smaller magnitudes indicating better performance. The box indicates the lower and upper quartile and median values. The whiskers show the extent of the rest of the data. Outliers are labeled as blue stars. Single lines are used when only one datum is available (U-Tree and human).

I do not show results for quadratic and SPOVA approximators. Preliminary tests showed that in almost all cases the quadratic model failed to converge to a good approximation. I was unable to fit the SPOVA approximator due to numerical instabilities. In all cases the parameters of the SPOVA approximation grew to infinity. I was unable to determine why this was happening. The SPOVA approximator has been successfully used in a factored setting by Rodriguez, Parr, and Koller [2000].

The MLP approximator for both the localist and distributed representation performed better than the linear approximation. Using ARD improved the DBN performance significantly. We can see that both localist and distributed models performed comparably to or better than U-Tree. However, it should be noted that U-Tree was only run for 30 000 time steps, while the other methods (except for the genetic algorithm) were allowed to

[5]Because testing time was shorter, the results were multiplied by 5/2.

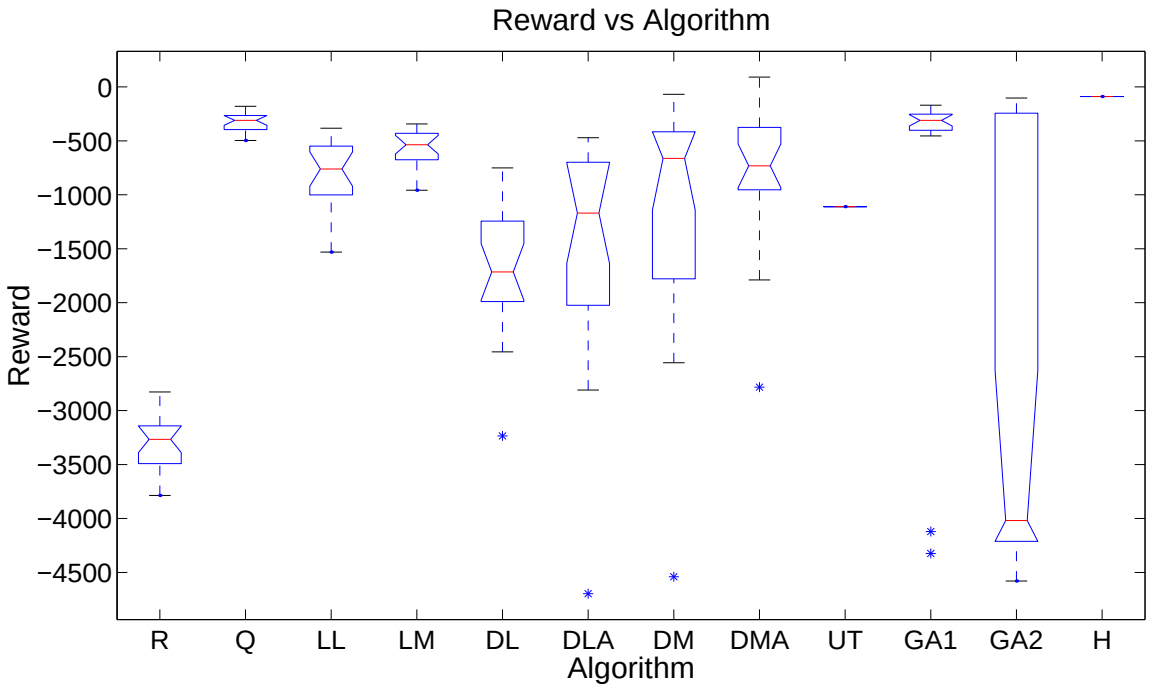

Figure 3.4: Results of algorithms on New York driving. R==Random driver; Q==flat Q-learning; LL==localist linear; LM==localist MLP; DL==distributed linear; DLA==distributed linear with ARD; DM==distributed MLP; DMA==distributed MLP with ARD; UT==U-Tree [McCallum 1997]; GA1==genetic algorithm, stochastic [Glickman and Sycara 2001]; GA2==genetic algorithm, sigmoid [Glickman and Sycara 2001]; H==human driver

train for 100 000 time steps. The variance of the DBN models is higher than the localist methods, and the DBN models occasionally converge to a poorly-performing solution. All of the models except for flat Q-learning perform worse than the stochastic GA algorithm, but the GA algorthm sometimes results in extremely poor solutions. Again, it should be noted that the GA algorithm required several orders of magnitude more training time than the other algorithms (5×10^8 steps in total). The most striking result is that flat Q-learning does so well. It equals or outperforms all of the more sophisticated algorithms and almost equals the human performance. It also consistently converges to a good solution. This indicates that the amount of "hidden state" in the original task is not very great. A memoryless (and deterministic) policy can do very well on this task.

One might ask what constitutes "good" performance on this task. Unfortunately

we do not know the optimal policy and so can not compare directly to optimal performance. We do have access to the underlying core states. If we could solve the underlying MDP directly, then we could at least get an upper bound on optimal performance.[6] Unfortunately the core state space is extremely large. I tried to approximately solve the underlying MDP with Q-learning using two different approximation methods. The first was simple state aggregation. Instead of distinguishing between 16 different distances from the driver, I just distinguished between four: in front or behind, and greater than or less than 20 meters away. I also ignored all colours. The second method was to use a set of hand-picked binary features f_i and learn an approximate linear action-value function by learning the weights w_{ia} of the features. The approximate value of a state/action was:

$$\widehat{Q}(s,a) = \sum_i f_i(s) w_{ia} \tag{3.54}$$

where $f_i(s)$ is one if the feature is satisfied in the current state, and zero otherwise. The features were:

1. Is a horn blowing? (1 feature)
2. Is a slow car in front of me in lane k at a distance from 0m to 8m? (4 features)
3. Is a slow car in front of me in lane k at a distance from 8m to 20m? (4 features)
4. Is a fast car behind me in lane k at a distance from 0m to 8m? (4 features)
5. Is a fast car behind me in lane k at a from 8m to 20m? (4 features)
6. Am I currently in lane k? (4 features)
7. Am I looking ahead to the left/center/right? (3 features)
8. Am I looking behind to the left/center/right? (3 features)

[6] Thanks to Bob Price for suggesting this approach.

There were a total of 27 features.

The two different algorithms were allowed to train for 1 000 000 iterations. Unfortunately neither algorithm performed as well as flat Q-learning on the visible observations. In the absense of a solution to the underlying MDP, I use human performance on this task as an estimate of "good" performance. The other information that we have about performance is that the maximum attainable reward occurs when there are no collisions or horns. If this is achievable, the learner would get a reward of 500 after 5000 steps. Unfortunately, it is not clear whether or not this reward is achievable by any policy.

3.6.2 Restricted NY Driving

In order to increase the amount of "hidden state" in the problem, I removed some of the sensory inputs. I found that the performance of the flat Q-learner was significantly degraded when a single input, "gaze side", was removed. Figure 3.5 shows the performance of a subset of the algorithms on this restricted version of the driving task. All parameters (learning rates, temperatures, training time, number of states, etc.) were unchanged. Again, results are measured on the last 5000 steps of each simulation. The flat Q-learner, while doing better than random, is nowhere near the performance level of the more sophisticated algorithms. The performance of the flat Q-learner does not improve if it uses a stochastic policy (implemented by halting learning at a temperature of 0.25 instead of 0.1). The localist algorithms have mixed performance. The reward achieved by the DBN-MLP algorithm is roughly the same as on the previous task. The human performance shown is performance on the original task. It is just included here for comparison. There is no reason to believe that the human performance would degrade after removing a single state variable.

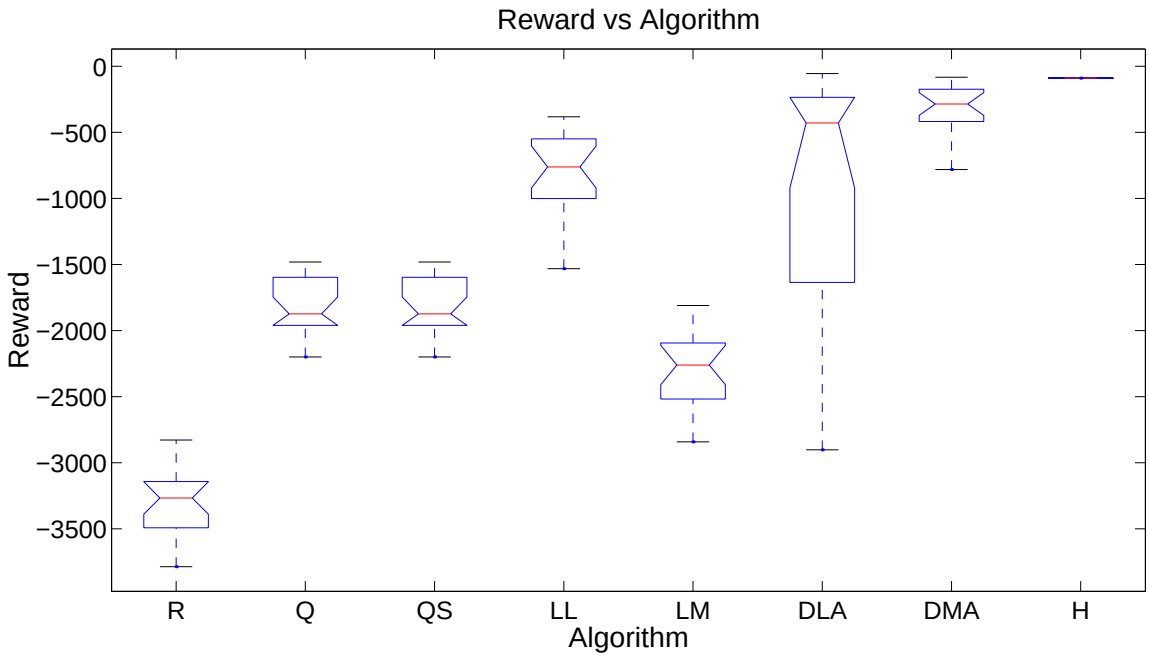

Figure 3.5: Results of algorithms on Restricted NY driving. R==Random driver; Q==flat Q-learning; QS==flat Q-learning, stochastic policy; LL==localist linear; LM==localist MLP; DLA==distributed linear with ARD; DMA==distributed MLP with ARD; H==human driver

3.6.3 Visual New York Driving

The New York Driving task uses relatively high-level sensory input. Attributes like "looming" and "receding" already provide a lot of information that is specific to solving the task. One nice aspect of the DBN approach is that it does not need to be given such high-level attributes. To demonstrate this I created the "visual" New York driving task.

The underlying task and reward structure are the same as in the original New York Driving task. But instead of using the sensory input shown in Table 3.1, the learner has access to an 8×8 image and a one-bit "Hear Horn" sensor. The 8×8 image was designed to convey instantaneous information such as gaze distance, gaze object and colour. The pixels of objects are coloured as in the original task. There is also a background colour (black), so each pixel can take on one of 7 values (meaning that there are approximately 2^{180} possible sensory inputs). Of course, only a tiny fraction of possible inputs are observed. Example images and the corresponding states are shown in figure

3.6. Notice that a lot of information such as "looming" or "gaze direction" is absent from the instantaneous input although it can be inferred by remembering previous states and actions. The idea is to force the learner to use memory to solve the task.

A DSBN with a 32-bit core state space was trained on the Visual New York Driving task. An MLP was used to approximate the value function, and ARD was used during model learning. The model was trained for 60 000 time steps. Localist models with 32, 128 and 512 states were trained for comparison. The localist models were allowed to train for 100 000 time steps. The learning rate and temperature schedule were unchanged. I also tried one run of the localist model with ARD for comparison purposes.

Finally, because very few images are actually seen by the learner (only 78 out of approximately 2^{180}) one might wonder if dimensionality-reduction could be performed as a pre-processing step, followed by memoryless Q-learning. I enumerated the possible sensory inputs (there are 78 of them) and used these as the states of a table-based Q-learner. The result of running on this "reduced" input is also shown. I also tested the best localist algorithm and the DBN algorithm on the reduced input.

Each algorithm was run five times. The total reward over the last 5000 steps was averaged across the five runs. Performance is shown in figure 3.7.

The localist model does increasingly well as the state space grows, although this improvement is not statistically significant. The table-based Q-learner trained on the reduced input does about as well as the table-based Q-learner on the restricted NY driving task, and better than the localist models. All are inferior to the DBN method. The DBN method has one bad run where the performance is comparable to flat Q-learning. The other runs are significantly better than the table-based Q-learner. The DBN also has performance similar to that on the previous versions of the task: original and restricted New York Driving. Again, note that the human performance is measured on the original task. It is only included here for comparison purposes. A human driver might not necessarily do so well on the visual version of the task.

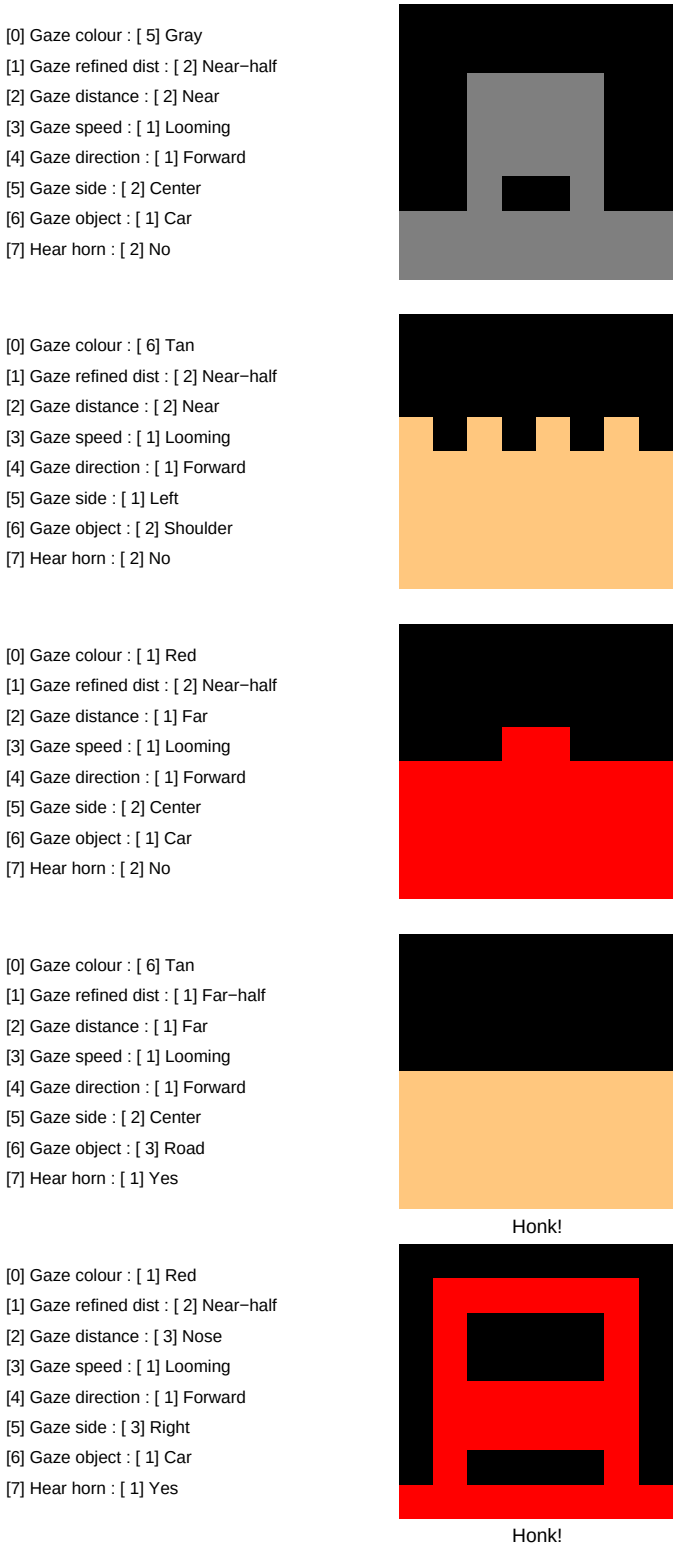

Figure 3.6: Examples of New York Driving state and the associated image from Visual New York Driving

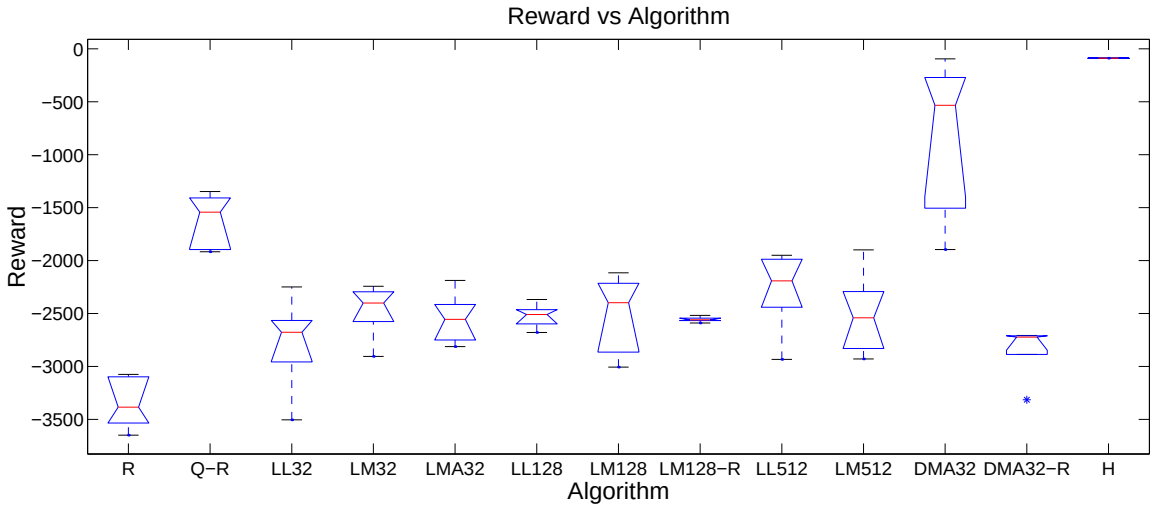

Figure 3.7: Results of algorithms on Visual NY driving. R==Random driver; Q-R==table Q-learning, reduced input ; LL32,LL128,LL512==localist linear (32,128,512 states); LM32,LM128,LM512==localist MLP (32,128,512 states); LM128-R==localist MLP, reduced input (128 states); LMA32==localist MLP with ARD (32 states); DMA32==distributed MLP with ARD (32 hidden variables); DMA32-R==DBN, MLP, ARD, reduced input (32 hidden variables); H==human driver

Interestingly both the localist and the DBN model do poorly on the reduced input. This would suggest that the structure in the images is very important to the success of the algorithm. The fact that there are strong correlations between input variables at the same time step, as well as across time steps, helps these models tremendously. Without these strong correlations it is harder for the models to learn about the structure of the time series data from their noisy gradient signals.

Figures 3.8–3.12 show a typical sequence involving horn and collision avoidance. The image on the left is a bird's-eye view of the road. The sequence of three images on the right shows the actual state perceived by the agent. Note that "forward" is toward the top of the image. Slow cars appear at the top of the image and disappear off the bottom.

Figure 3.13 shows emission features learned by the model. The figure was generated by turning on each binary unit and generating an image. Colour has been ignored in these images. Binary units without significant weights to the image have been omitted. The learner models cars at various distances. It also uses one bit in its core state to

indicate that a horn is being honked behind the driver. From watching the learner in action, it is clear that it also models aspects of its internal state, such as gaze side. This allows it to quickly look right (when it sees the red car, figure 3.10) and then left, even though the previous gaze side is not immediately available from the state. See section 5.1 for more on interpreting the behaviour of hidden units in the model.

3.6.4 Eligibility Traces

So far I have only discussed final performance for each algorithm, and ignored the rate at which the methods learn the task. Here I experiment with eligibility traces to investigate the effect of varying λ on the rate of convergence to a good solution. I use a 16-unit DSBN with an MLP function approximator and ARD. The task is restricted NY driving. Figures 3.14 and 3.15 show the effect of varying λ on speed of convergence for two types of eligibility traces: Sutton and Barto's "naive" trace, and Watkins' trace. Both traces use the general eligibility trace form from Eq.(3.48). Recall that as λ moves from zero to one, the method comes closer to being a full Monte Carlo method. The lower line is at a value of -222, which is the reported performance of the U-Tree algorithm. The upper line is at a value of -49.1, which is the reported performance of the genetic algorithm with a stochastic recurrent network.

The performance of the current greedy policy was found by holding all parameters fixed and simulating for 1000 iterations.[7] Each iteration was comprised of updating the state, selecting an action, and modifying parameters. The figures show performance averaged over 20 learning trials for each value of λ.

Convergence to a good solution is fastest for intermediate values of λ. In particular, both methods seem to work best for either $\lambda = 0.2$ or $\lambda = 0.4$. Watkins' method with $\lambda = 0.4$ appears to work best overall, with fastest convergence to a good solution and

[7]The evaluation trial used a fixed random seed, so that it always had the same initialization and sequence of obstacle cars. After the policy was evaluated the random seed was restored to the value it held just prior to the evaluation trial.

lowest variance across solutions. Both algorithms failed to converge to a good solution when $\lambda = 1$. These results are consistent with previous results on learning with eligibility traces [Singh and Sutton 1996; Singh and Dayan 1998; He and Shayman 1999], although the specific best value for λ varies for different problems in the literature. Note that for $\lambda = 0$, both traces reduce to the original learning algorithm investigated in section 3.6.2.

At its best, the DBN algorithm learns the task in about the same number of iterations as McCallum's U-Tree (which was trained for 30 000 actions), and had performance comparable to the genetic algorithm.

Table 3.2: Algorithms Tested with the New York driving task.

Algorithm	Code	Model	Regularization	Inference	Value Funtion
Random	R	none	none	none	none
Flat Q-learning	Q	enumeration of state-actions	none	none	Table
Localist linear	LL	hidden Markov model	none	exact	linear
Localist MLP	LM	hidden Markov model	none	exact	MLP
Distributed linear	DL	DSBN	none	mean-field	linear
Distributed Linear with ARD	DLA	DSBN	ARD	mean-field	linear
Ditributed MLP	DM	DSBN	none	mean-field	MLP
Distributed MLP with ARD	DMA	DSBN	ARD	mean-field	MLP
U-Tree	UT	decision tree	none	exact	linear
Genetic algorithm, stochastic	GA1	stochastic recurrent network	none	none	none
Genetic algorithm, sigmoid	GA2	sigmoid recurrent network	none	none	none
Human	H	unknown	unknown	unknown	unknown

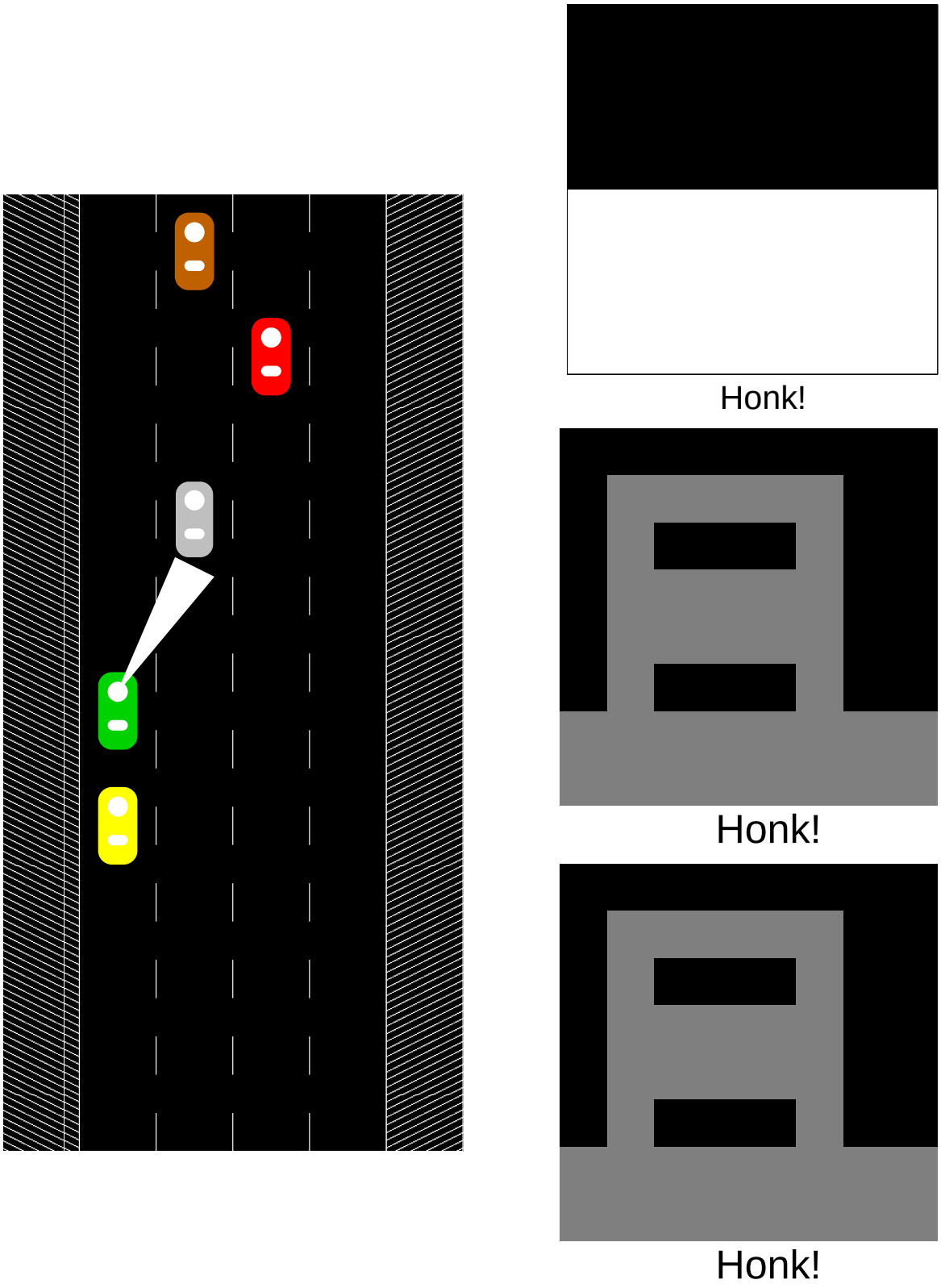

Figure 3.8: The agent (green car) has a clear road, but there is a fast car (yellow) blowing its horn behind. The agent looks to the right, but sees a looming gray car. The agent's gaze direction is symbolized by the white cone.

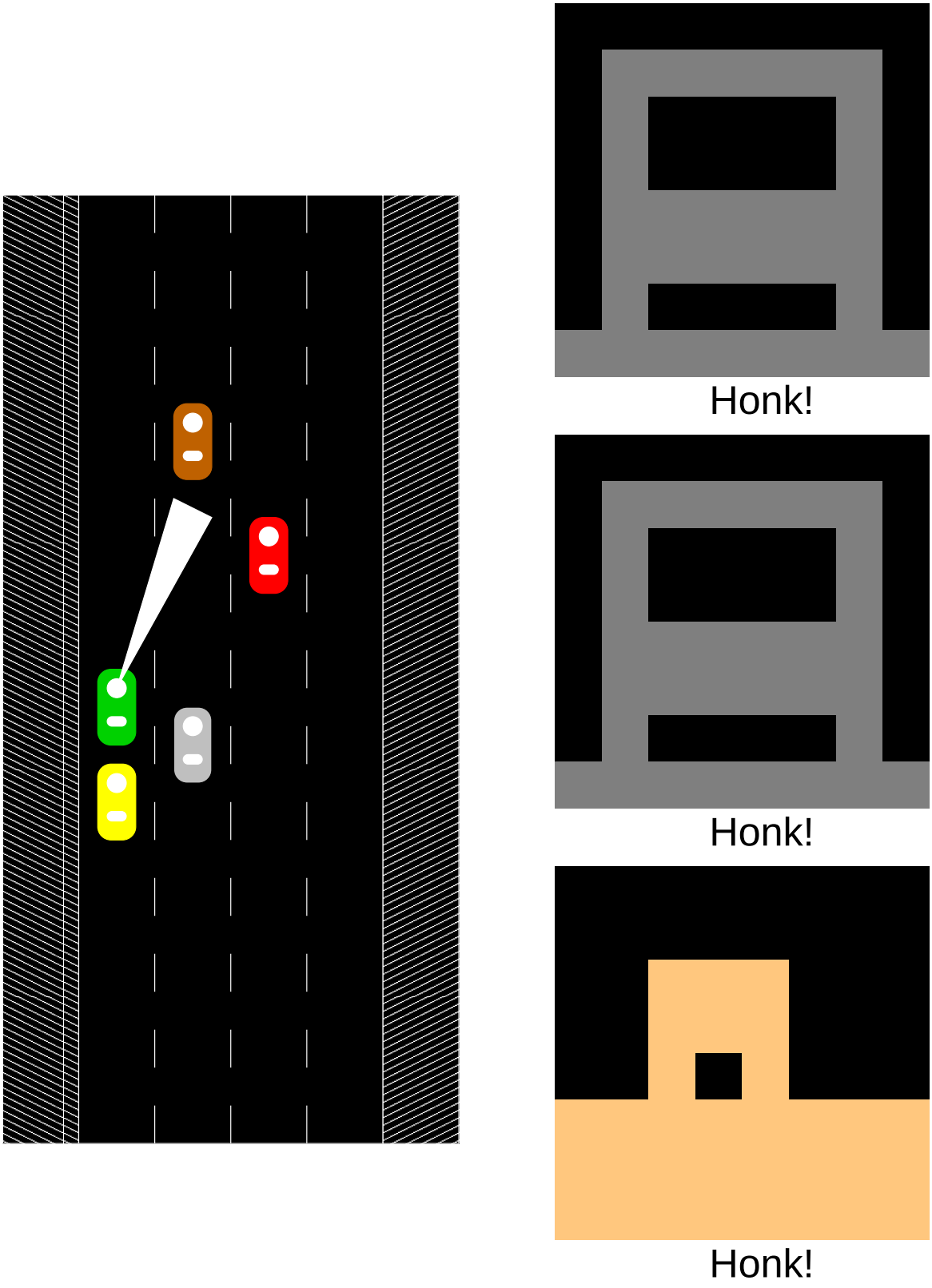

Figure 3.9: The agent passes the gray car, and sees a tan car in the same lane.

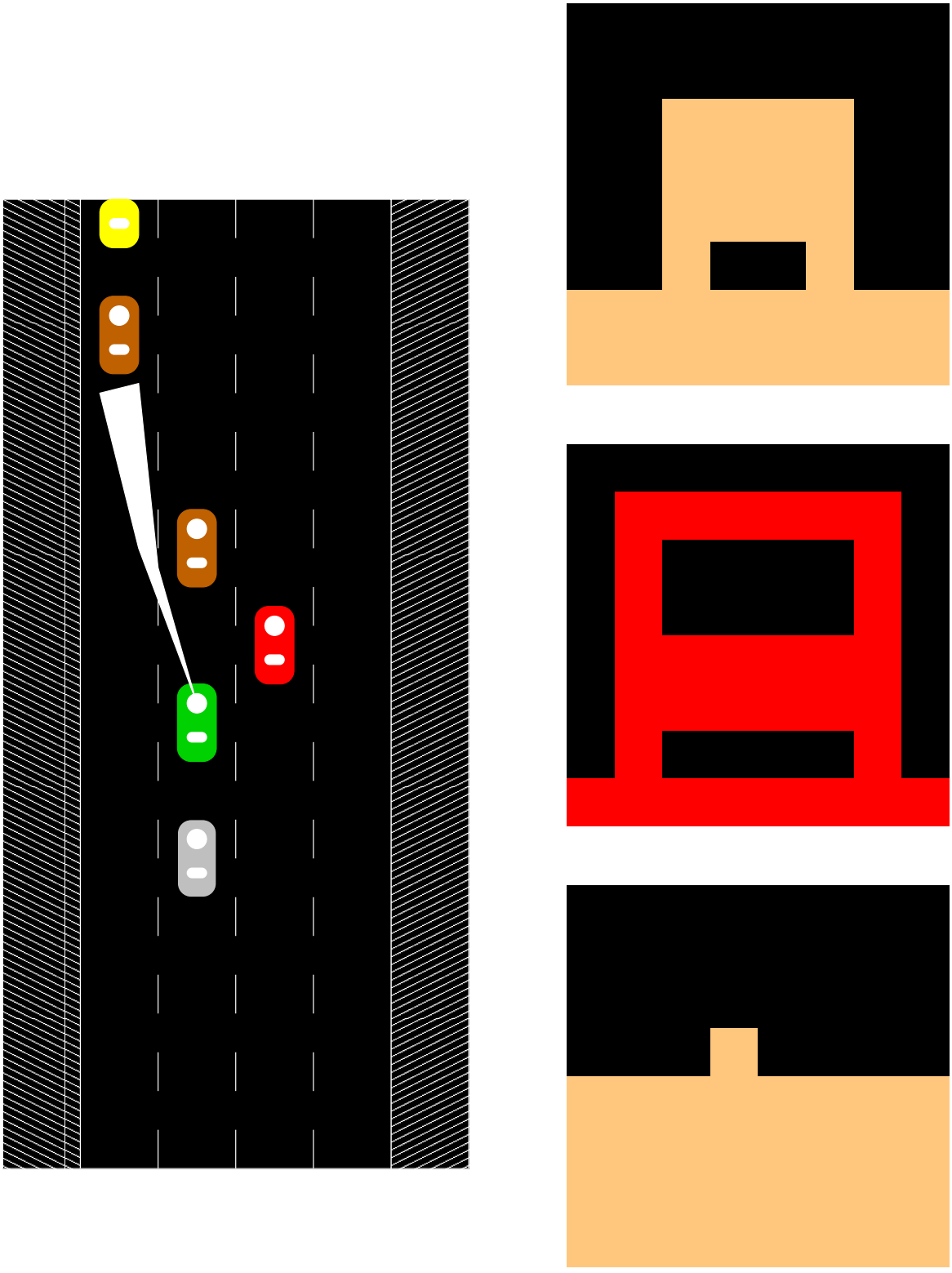

Figure 3.10: The agent changes lanes to the right, moving behind the tan car. This lets the fast yellow car speed away (off the top of the display). Now the tan car is looming ahead. The agent looks to the right and sees a nearby red car. It looks to the left and sees a far tan car.

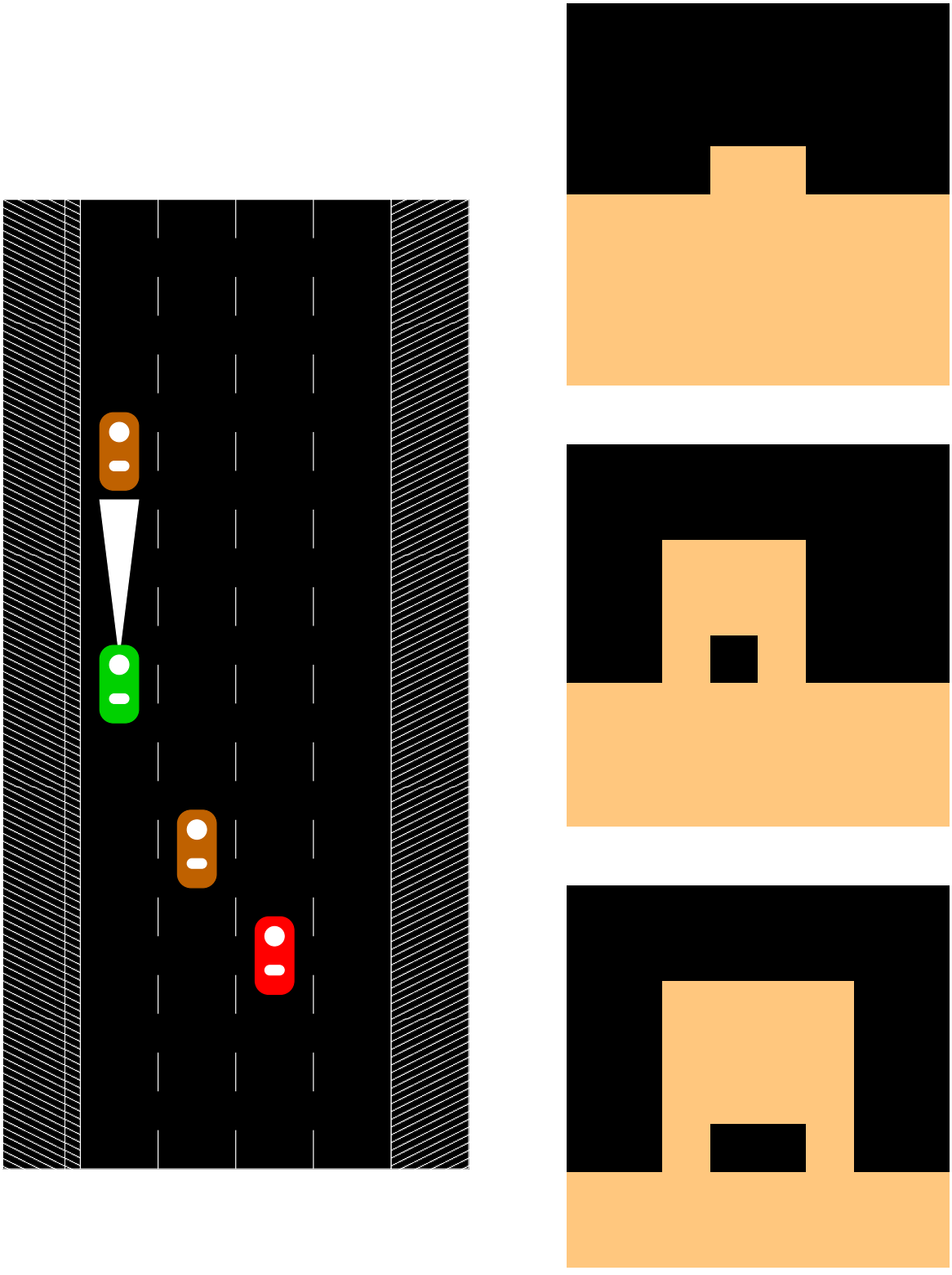

Figure 3.11: The agent changes lanes to the left, moving behind the tan car. The other slow cars are left behind (drop off the bottom of the display). The tan car ahead looms closer.

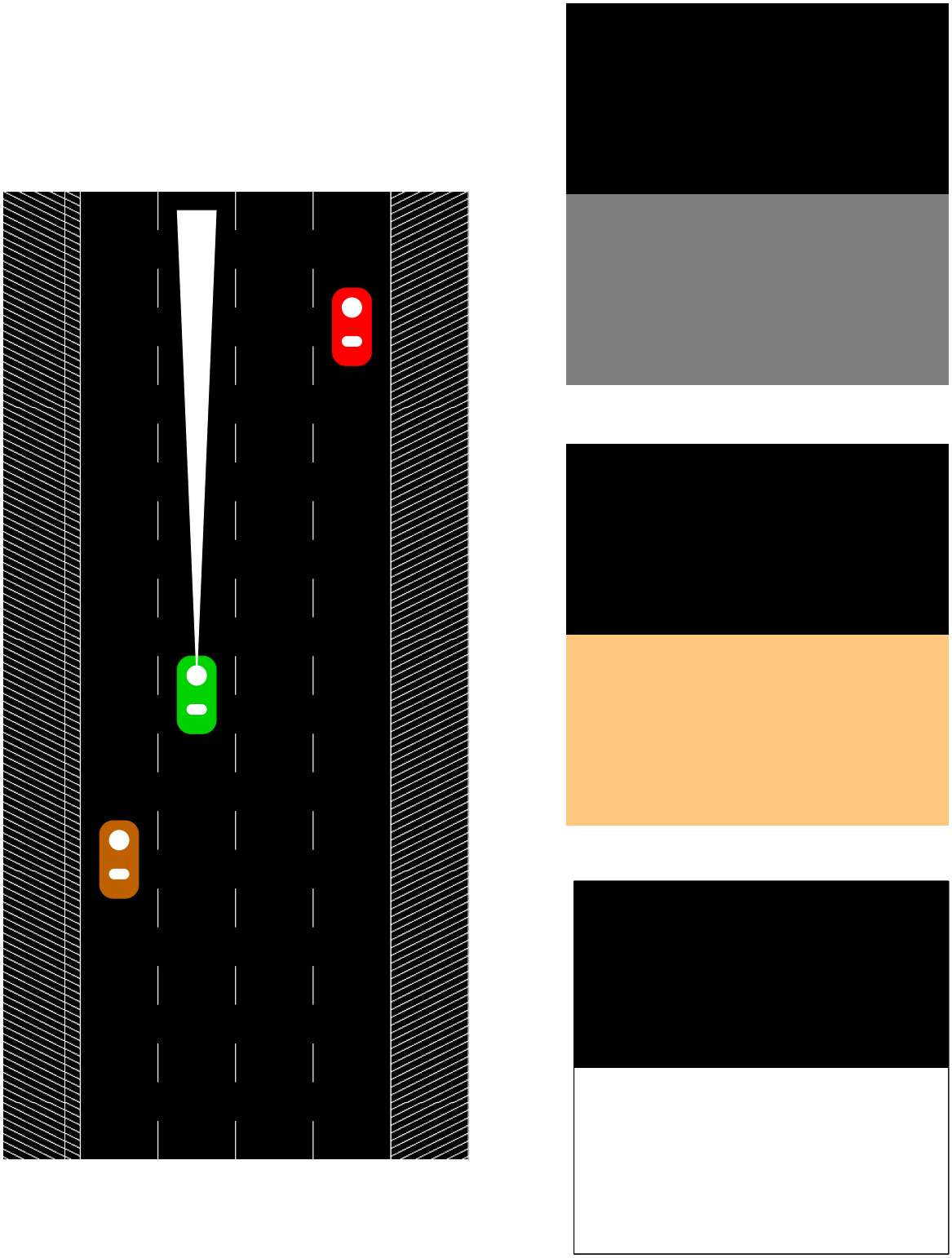

Figure 3.12: The agent looks to the right, and sees a free lane. It shifts to the right, passing the looming tan car.

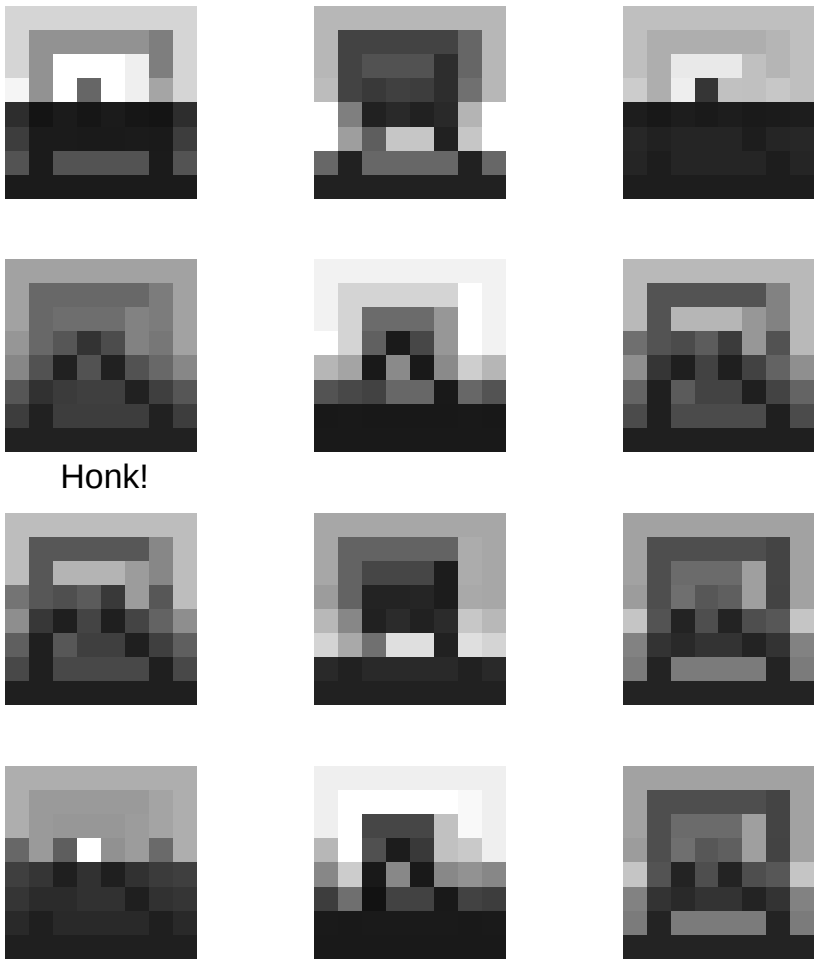

Figure 3.13: Emission features learned for Visual New York Driving. Each feature corresponds to what makes a binary hidden unit active. Binary units with no relevant output have been omitted. White indicates higher and black indicates lower probability of activity in the associated binary unit.
For example, feature (2,2) becomes progressively more active as a car gets closer (small car outlines are dark, bigger car outlines get lighter). Feature (4,1) is very active for cars at the maximum distance (small white dot), but is inactive for "shoulder" images (the other small gray dots on the horizon). Feature (2,1) is not very interested in cars, but is very active when a horn is honking behind the agent.

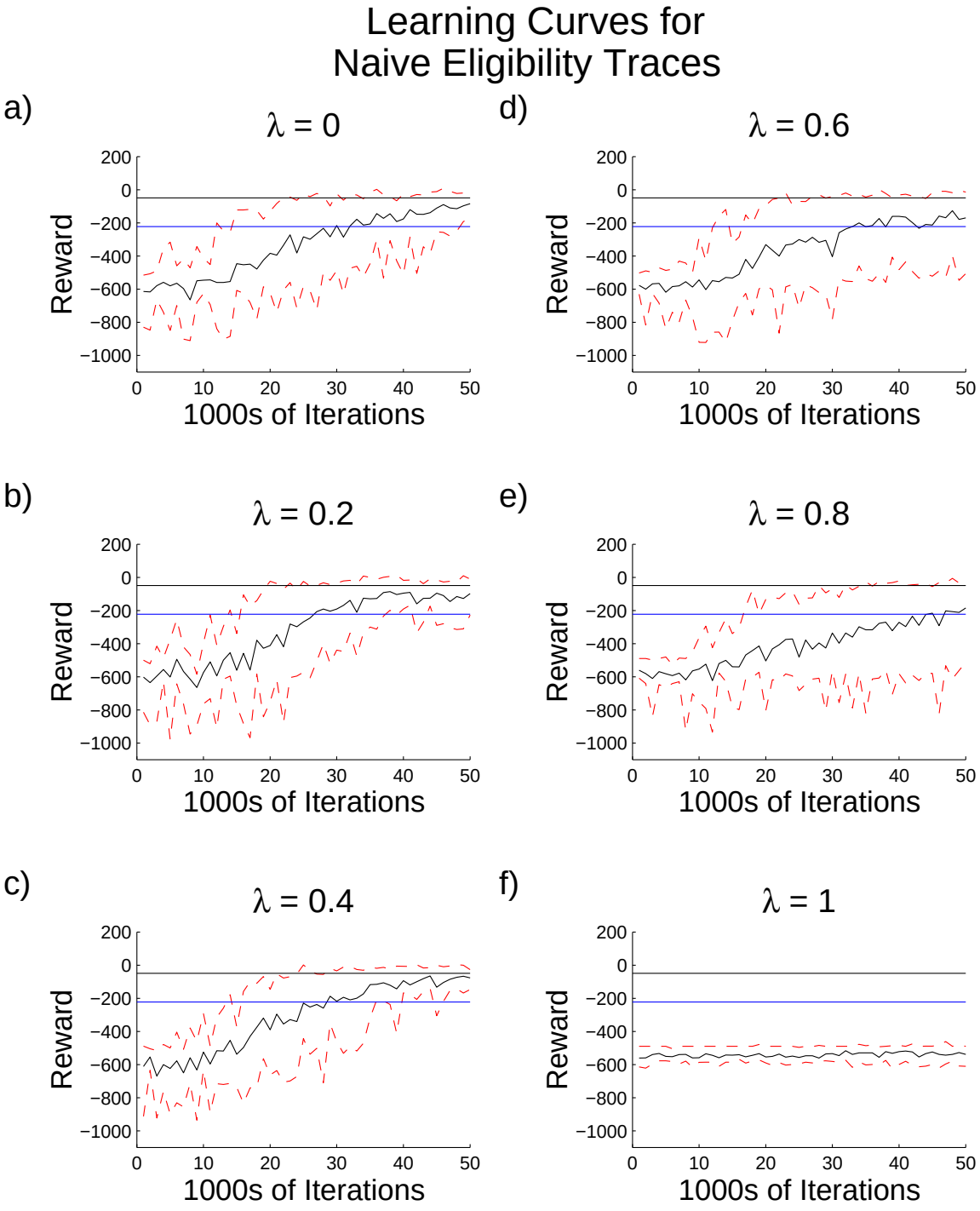

Figure 3.14: Performance of Naive traces as a function of λ for restricted NY driving. The center line on each plot shows the mean reward across 20 trials. The dashed lines contain 80% of the reward values across all 20 trials. The straight lines show the reported final performance of McCallum's U-Tree algorithm (lower) and of Glickman and Sycara's genetic algorithm (upper).

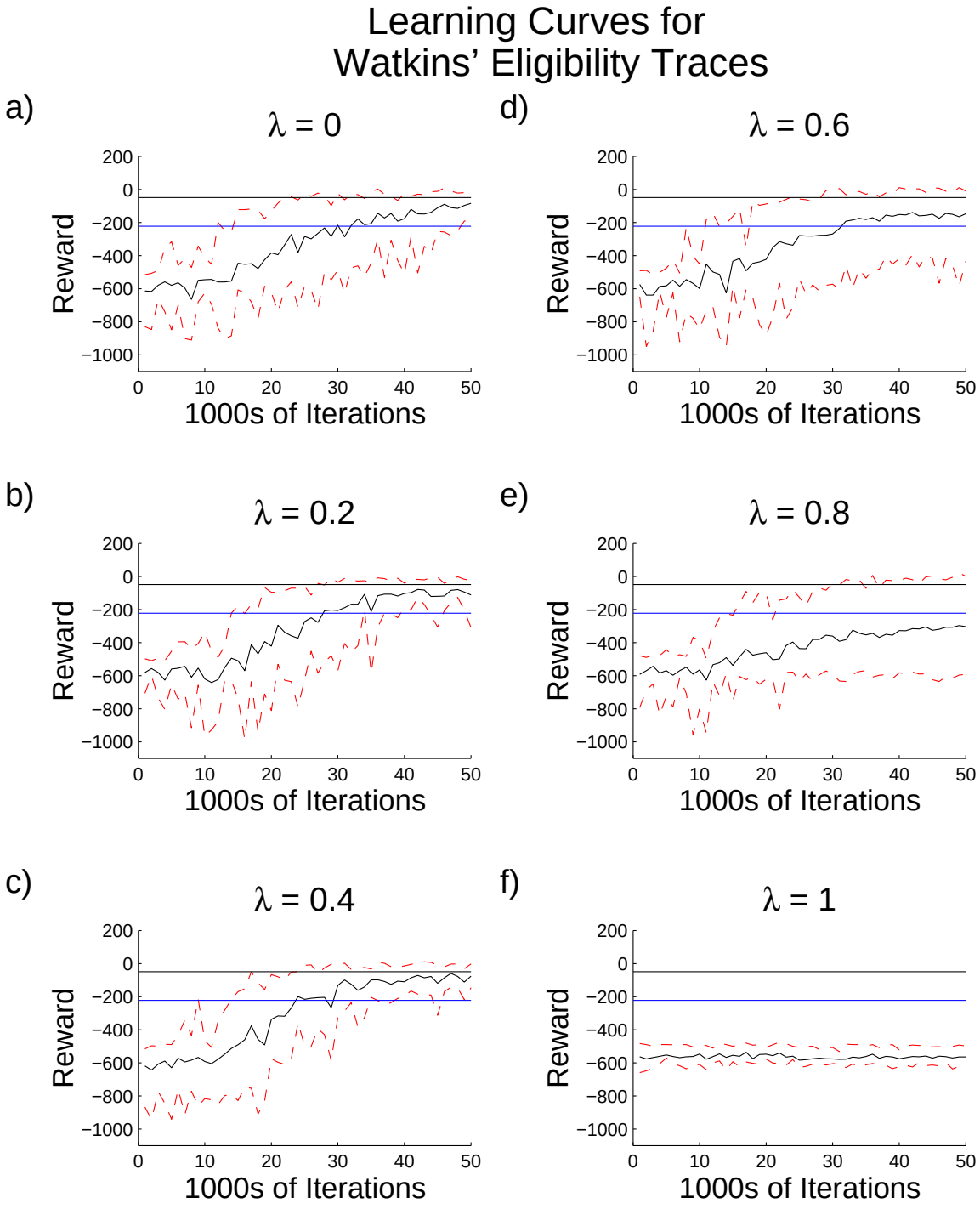

Figure 3.15: Performance of Watkins' traces as a function of λ for restricted NY driving. The center line on each plot shows the mean reward across 20 trials. The dashed lines contain 80% of the reward values across all 20 trials. The straight lines show the reported final performance of McCallum's U-Tree algorithm (lower) and of Glickman and Sycara's genetic algorithm (upper).

Chapter 4

Factored Actions

In this chapter, I present a new method for the compact representation of action value functions of high-dimensional factored MDPs. The representation allows for the efficient selection of good actions from a high-dimensional action space based on their estimated value. Initial results of this work were presented in [Sallans and Hinton 2001].

There has been a lot of research and discussion about the problems of dealing with large state spaces. There has been comparatively little on dealing with large action spaces [Dean, Givan, and Kim 1998; Meuleau, Hauskrecht, Kim, Peshkin, Kaelbling, Dean, and Boutilier 1998; Peshkin, Meuleau, and Kaelbling 1999; Guestrin, Koller, and Parr 2002]. The action space might be large because the action is actually composed of many action variables. It also might be infinite if the action is real-valued. In practice action spaces tend to be very large. For example consider the activation of large numbers of muscles, or the simultaneous actions of all of the players on a football field.

In previous chapters I have discussed table-based and function-approximator based reinforcement learning. We have seen that in many cases a table-based representation of the value function is not appropriate. This is particularly true of factored MDPs where the size of the state or action space grows exponentially with the number of state or action variables. Similarly, if the state or action space is continuous, using a table-based

representation is usually inappropriate.

If a non-linear function approximator is used to model the action-value function, then extracting the policy involves solving a constrained non-linear optimization problem for each action selection. In this chapter I present a method whereby the value of a state-action pair is represented by the negative free energy, $-F$, of the state-action pair under a non-causal graphical model. Because of this interpretation of the action-value as negative free energy it will be possible to select good actions (although not necessarily optimal actions) efficiently.

4.1 Stochastic Policies

When trying to solve a POMDP, an often-used technique is to solve it as if it were a fully observable MDP. The MDP framework has often been applied to problems that are not strictly Markov in the observable state. Policies that only use the currently visible state to select actions are known as "memoryless" policies. It has been shown that stochastic memoryless policies can perform better on POMDPs than deterministic memoryless policies [Jaakkola, Singh, and Jordan 1995]. With this result in mind, there has been recent work on learning stochastic policies with either value-function or direct-policy based methods. Direct-policy methods have been extended to include memory by adding additional "memory bits" which can be turned on and off, and which become part of the state at the next time step [Peshkin, Meuleau, and Kaelbling 1999]. Turning memory bits on and off can be formalized by extending the action set to include actions that flip these bits. This is another case where the action space can quickly become very large.

All of these methods make strict assumptions about the structure of the stochastic policies that can be represented. For example, a value-function based method might map

from state-action pairs to values, and use a Boltzmann distribution to express the policy:

$$P(a|s) = \frac{\exp\{Q(s,a)\}}{\sum_{a'=1}^{A} \exp\{Q(s,a')\}} \tag{4.1}$$

where the denominator is a sum over all possible actions. A direct policy method might map from states to the probability of performing each action. Both cases prove problematic for high-dimensional factored actions. The summation over all actions in the value-function method will quickly become intractable. And in the case of parameterized policies, a family of parameterized policies must be chosen in advance.

One possible choice is the family of factored policies [Peshkin, Meuleau, and Kaelbling 1999]. The probability of selecting each element of the action is independent of the other elements given the current state. Another choice is policies where some subsets of the action variables are correlated [Guestrin, Koller, and Parr 2002]. However, the details of the distribution: which action variables are correlated, how many modes the conditional action distribution has, and so on, must all be pre-specified.

To illustrate this point, consider a problem with a two-bit action, where the optimal policy is shown in Table 4.1. If the family of policies that we pre-selected for solving

Table 4.1: Optimal Stochastic Policy with No Factored Representation

Action	**Probability**
0 0	0
0 1	1/2
1 0	1/2
1 1	0

this problem were the family factored policies, then we would not be able to find a good policy in this case. Of course, given prior knowledge of the problem we might be able to select a family of parameterized policies in which a good policy exists. But this prior knowledge of the problem might be hard to come by if all we can do is draw sample paths

according to the MDP. I will return to this example in section 4.4.1.

4.2 Products of Experts

I will represent the value of a state-action pair as the free energy of the pair under a probabilistic model. The model that I will use to represent the value function is a product of experts [Hinton 2000]. Products of experts are probabilistic models that combine simpler models by multiplying their distributions together.

With a product of experts , the probability assigned to a state-action pair, $(\mathbf{s}, \mathbf{a})$, is the (normalized) product of the probabilities assigned to $(\mathbf{s}, \mathbf{a})$ under each of the individual experts:

$$P(\mathbf{s}, \mathbf{a}|\theta_1, ..., \theta_K) = \frac{\prod_{k=1}^{K} P_k(\mathbf{s}, \mathbf{a}|\theta_k)}{\sum_{(\mathbf{s}', \mathbf{a}')} \prod_k P_k(\mathbf{s}', \mathbf{a}'|\theta_k)} \tag{4.2}$$

where P_k is the probability distribution of expert k; $\{\theta_1, ..., \theta_K\}$ are parameters of the K experts; and $(\mathbf{s}', \mathbf{a}')$ indexes all possible state-action pairs.

A product of experts model can be defined by an energy function which defines a joint probability distribution:

$$E(\mathbf{s}, \mathbf{a}|\theta_1, ..., \theta_K) = \sum_{k=1}^{K} E_k(\mathbf{s}, \mathbf{a}|\theta_k) \tag{4.3}$$

The energy is equal to the negative log probability up to an additive constant. The constant is equal to the log of the partition function:

$$p(\mathbf{s}, \mathbf{a}|\theta_1, ..., \theta_K) = \frac{\exp\{-E(\mathbf{s}, \mathbf{a}|\theta_1, ..., \theta_K)\}}{\sum_{(s', a')} \exp\{-E(\mathbf{s}', \mathbf{a}'|\theta_1, ..., \theta_K)\}} \tag{4.4}$$

where the denominator is called the partition function. Computing the partition function will typically be intractable for large state-action spaces. However, the energy is easily computed.

If there are additional latent variables in the model, then the distribution over the

observed variables can be defined in terms of the free energy:

$$F(\mathbf{s}, \mathbf{a}|\theta_1, ..., \theta_K) = \sum_{k=1}^{K} \langle E_k(\mathbf{s}, \mathbf{a}, \mathbf{h}_k|\theta_k)\rangle_{P(\mathbf{h}|\mathbf{s},\mathbf{a})} - H(P(\mathbf{h}|\mathbf{s}, \mathbf{a})) \tag{4.5}$$

where $\mathbf{h}$ denotes a vector of hidden variables. The first term is the expected energy, and the second term is the entropy of the posterior distribution over hidden units. For Bayesian networks the partition function is unity, so the energy is equal to the negative log probability of visible and hidden variables. For product models, which are undirected, the equality only holds up to an additive constant.

The free energy has two terms: The expected energy with respect to the posterior distribution of the hidden variables given the visible variables; and the entropy of the posterior. The posterior distribution is given by:

$$\begin{aligned} P(\mathbf{h}|\mathbf{s}, \mathbf{a}) &= \frac{\prod_{k=1}^{K} \exp\{-E_k(\mathbf{s}, \mathbf{a}, \mathbf{h}_k|\theta_k)\}}{\sum_{\mathbf{h}'} \prod_{k=1}^{K} \exp\{-E_k(\mathbf{s}, \mathbf{a}, \mathbf{h}'_k|\theta_k)\}} \\ &= \prod_{k=1}^{K} \frac{\exp\{-E_k(\mathbf{s}, \mathbf{a}, \mathbf{h}_k|\theta_k)\}}{\sum_{\mathbf{h}'_k} \exp\{-E_k(\mathbf{s}, \mathbf{a}, \mathbf{h}'_k|\theta_k)\}} \\ &= \prod_{k=1}^{K} P(\mathbf{h}_k|\mathbf{s}, \mathbf{a}) \end{aligned} \tag{4.6}$$

So the posterior distribution of a product of experts factors into the product of the posterior distributions for each of the individual experts. As long as inference in each of the individual experts is tractable, inference is tractable in the product model.

If inference is tractable, then the product of experts has two very useful properties. First, given a state-action pair the exact free energy is easily computed. Second, the derivative of the free energy with respect to each parameter of the network is also simple. These two properties will be vital for using products of experts for reinforcement learning.

Examples of product models include restricted Boltzmann machines [Smolensky 1986; Hinton 1999], products of hidden Markov models [Brown and Hinton 2001], products of

Gaussian mixtures [Hinton 1999], and rate-coded restricted Boltzmann machines [Teh and Hinton 2001]. Again, notice that in each case the individual experts (binary logistic units, hidden Markov models and Gaussian mixtures) are themselves tractable.

The model is trained to minimize the temporal-difference error using a SARSA or Q-learning update rule (see section 4.6). After training, a good action for a given state can be found by clamping the state and drawing a sample of the action variables using Gibbs sampling [Geman and Geman 1984]. Although finding optimal actions would still be difficult for large problems, selecting an action with a probability that is approximately proportional to $\exp(-F)$ can normally be done with a small number of iterations of Gibbs sampling (sometimes including simulated annealing). In principle, if we let the Gibbs sampling converge to the equilibrium distribution, we could draw unbiased samples from the Boltzmann exploration policy at a given temperature. In particular we can select actions according to a policy that may be much too complicated to parameterize explicitly. Also, we do not have to decide *a priori* how many modes the policy distribution should have, which action variables may be correlated and so on. This method allows us to represent and access a much larger class of policies than was previously possible using either direct-policy or actor-critic methods. In particular it allows us to represent policies that have no compact functional representation. It also allows us to learn stochastic policies when we do not know a priori what correlations should be represented (see sections 4.4.1 and 4.8.1). It also provides a way of selecting approximately optimal actions in large discrete and real-valued action spaces.

4.3 Tractable Models and Bayesian Networks

We do not have to use a product of experts model to model the action-value function. Any probabilistic model would suffice. For example, we could use a completely tractable model such as a mixture of Gaussians or a singly-connected Bayesian network. In these

cases the partition function can be computed tractably, so we can sample actions from the model exactly, without resorting to Gibbs sampling.

Alternatively we could use a different class of models combined with approximate inference. For example, we could use multiply-connected Bayesian networks and a variational approximation. In this case, the value of a state-action pair would be represented by the (negative) *variational* free energy. Action selection would still involve an iterative procedure (the fitting of the variational parameters at the current state), but we would have access to the entire policy distribution $p(\mathbf{a}|\mathbf{s})$. This would be very closely related to actor-critic methods, and will be discussed further in section 5.2.

4.4 Discrete States and Actions

Many kinds of "experts" could be used while still retaining the useful properties of the PoE. In this section, I will focus on the case where each expert is a single binary logistic unit. This case is interesting because inference and learning in this model has been intensely studied [Ackley, Hinton, and Sejnowski 1985; Hinton and Sejnowski 1986; Smolensky 1986; Freund and Haussler 1992; Hinton 2000], and the binary values are relatively easy to interpret. Each (multinomial) state or action variable is represented using a "one-of-N" set of binary units which are constrained so that exactly one of them is on. The product of experts is then a bipartite "Restricted Boltzmann Machine" (RBM) [Smolensky 1986; Hinton 2000] (see figure 4.1(a)). I use s_i to denote the i^{th} state variable and a_j to denote the j^{th} action variable. I will denote the binary latent variables of the "experts" by h_k (see figure 4.1(b)). In the following, keep in mind that states are always clamped, and the actions are always sampled such that the one-of-N multinomial restrictions are respected. Given these restrictions, we can ignore the fact that the binary state vector actually represents the values of a set of multinomial state variables. Likewise for the binary action variable. The representation of the free energy

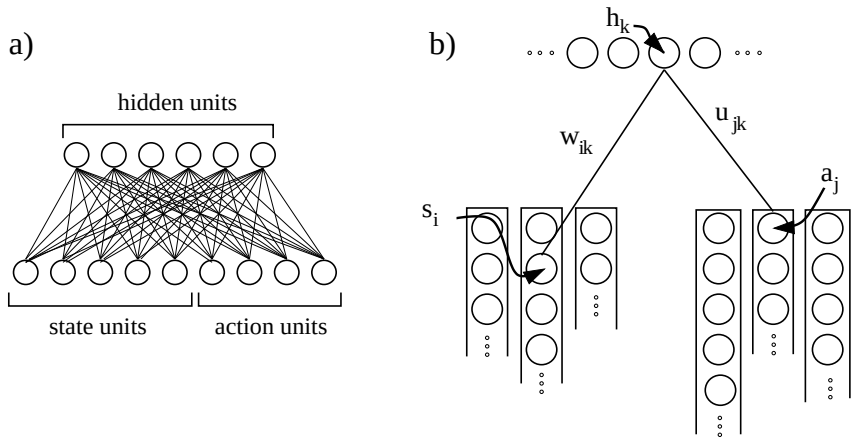

Figure 4.1: a) The Boltzmann product of experts. The estimated action-value of a setting of the state and action units is found by holding these units fixed and computing the free energy of the network. Actions are selected by holding the state variables fixed and sampling from the action variables. b) A multinomial state or action variable is represented by a set of "one-of-n" binary units in which exactly one is on.

is the same as in the binary case.[1]

For a state $\mathbf{s} = \{s_i\}$ and an action $\mathbf{a} = \{a_j\}$, the free energy is given by the expected energy under the posterior distribution of the hidden units minus the entropy of the posterior. The posterior distribution over the hidden units factors: The posterior distribution over each latent binary variable is independent of the others given the state and action. The free energy is simple to compute because the hidden units are independent in the posterior distribution:

$$\begin{aligned} F(\mathbf{s},\mathbf{a}) &= -\sum_{k=1}^{K}\left(\sum_{i=1}^{|\mathcal{S}|}(w_{ik}s_i\widehat{h}_k + b_i s_i) + \sum_{j=1}^{|\mathcal{A}|}(u_{jk}a_j\widehat{h}_k + b_j a_j)\right) - \sum_{k=1}^{K} b_k\widehat{h}_k \\ &\quad + \sum_{k=1}^{K}\widehat{h}_k \log\widehat{h}_k + (1-\widehat{h}_k)\log(1-\widehat{h}_k) \end{aligned} \tag{4.7}$$

where w_{ik} is the weight from the k^{th} expert to binary state variable s_i; u_{jk} is the weight

[1] Equivalently we can just think in terms of multinomial units or a Potts model.

from the k^{th} expert to binary action variable a_j; b_k, b_i and b_j are biases; and

$$\widehat{h}_k = \sigma\left\{\left(\sum_{i=1}^{|\mathcal{S}|} w_{ik}s_i + \sum_{j=1}^{|\mathcal{A}|} u_{jk}a_j\right) + b_k\right\} \tag{4.8}$$

is the expected value of each expert's latent variable given the data, where $\sigma(x) = 1/(1+e^{-x})$ denotes the logistic function. The first two terms of Eq.(4.7) correspond to an expected energy, and the third to the negative entropy of the distribution over the hidden units given the data. Again I emphasize that free energy can be computed tractably because inference is tractable in a product of experts.

4.4.1 Non-factored Policies

Recall the simple problem mentioned in section 4.1. It is clear that with a restricted Boltzmann machine and action sampling, the optimal policy can be represented. This is shown in figure 4.2. The hidden unit is on half of the time, and is off half of the time.

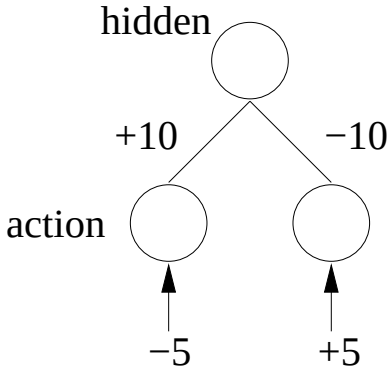

Figure 4.2: An illustration of the restricted Boltzmann machine that can represent a simple non-factored stochastic policy.

When it is on, action $\{1\ 0\}$ is selected. When it is off, action $\{0\ 1\}$ is selected.

The important point here is that a large number of different correlations between action variables can be captured by the value function approximator, and manifested in the policy during action sampling. We do not need to know ahead of time how many or which action variables are correlated, or how many modes are in the policy distribution. This can be learned by the model. The cost is that the actions must be sampled using

an iterative procedure.

4.5 Continuous States and Actions

We can also use free energies to model values over continuous-valued state and action spaces. In section 4.8.4 I will discuss a particular real-valued task. Here I describe two ways that a product model can be used to model continuous states and actions.

4.5.1 Rate coded RBM

Consider a restricted Boltzmann machine where each unit is duplicated n times. The total activity of each replicated set of units is an integer in the range $[0, n]$. We can approximate a real-valued unit by using a large number of duplicated discrete units. The "real" value will be able to take on one of $n + 1$ values.

One advantage is that the inference and learning rules for the duplicated units are unchanged, except that activities are now in the range $[0, n]$ instead of $[0, 1]$. Further, each replica of a unit makes the same computation of the probability of being on, so the computation can be performed simultaneously for all replicas. The sampling of (the sum of) activities given the probabilities of being on can be made efficient as well by the use of an approximation. Since the sum of independent binomial random variables converges to a normal distribution in the limit, we can approximate sampling from the sum of independent duplicate units by sampling from a Gaussian distribution with the appropriate mean and variance. If there are n replicas of a hidden unit, and the unit is active with probability p, then the mean is np, and the variance is $np(1 - p)$. Instead of selecting activities for the n independent units, we sample from a Gaussian with the same mean and variance, and round to the nearest integer.

Equivalently, we can think of a single unit that emits up to n spikes in a given time interval. The total activity of the ensemble of units is equal to the rate of spiking, with

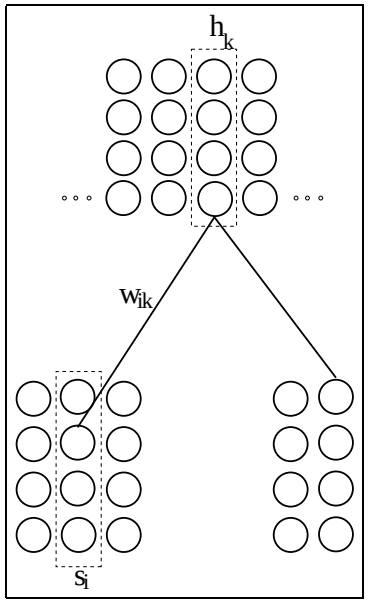

Figure 4.3: A rate-coded restricted Boltzmann machine. The real-valued activities of the units can be thought of as the total activity of an ensemble of duplicate units. Equivalently, we can think of a single unit that can emit a number of spikes in a given time interval. The real-valued activity is the rate at which the unit fires.

a maximum rate of n. For this reason this type of RBM is called a rate-coded restricted Boltzmann machine, or RBMrate [Teh and Hinton 2001].

4.5.2 Product of UniGauss

An alternative for real-valued states and actions is to use a product of simple mixture models. The uniGauss [Hinton 1999] is a mixture of a Gaussian and a uniform distribution. The latent variable is a discrete binary variable that indicates whether the Gaussian or uniform is active. In energy terms, if the Gaussian is active, then the uniGauss contributes quadratic energy around its mean. If the uniform is active, then the uniGauss contributes zero energy. The final energy given the hidden unit activities is the sum of the quadratic energies of the Gaussians which are active.

The free energy of the UniGauss model is given by:

$$\begin{aligned} F(\mathbf{s},\mathbf{a}) &= \sum_k \widehat{h}_k \left(-b_k + \frac{1}{2}\log 2\pi + \frac{1}{2}\log v_k + \frac{\left\| [\mathbf{s} \;\; \mathbf{a}]^\top - \mu_k \right\|_2}{2v_k^2} \right) \\ &\quad + \sum_k \left(\widehat{h}_k \log(\widehat{h}_k) + (1-\widehat{h}_k)\log(1-\widehat{h}_k) \right) \end{aligned} \tag{4.9}$$

where k indexes an element of the product model; D is the dimensionality of the vector $[\mathbf{s} \;\; \mathbf{a}]^\top$; b_k is a bias on the switching probabilities; μ_k and v_k are the mean and standard deviation of Gaussian k; and $\widehat{h}_k$ is the posterior probability of making Gaussian k active:

$$\widehat{h}_k = \left(1 + \exp\left\{ -b_k + \frac{1}{2}\log 2\pi + \frac{1}{2}\log v_k + \frac{\left\| [\mathbf{s} \;\; \mathbf{a}]^\top - \mu_k \right\|_2}{2v_k^2} \right\} \right)^{-1} \tag{4.10}$$

Here I have presented a model with isotropic Gaussians.

One disadvantage of RBMrate, and an advantage of a uniGauss model, is that in the former a real value is encoded as uncertainty about the state of a binary hidden unit. For the real value to take on intermediate values, the distribution of the hidden unit must have high entropy. The model cannot describe the case: "I am very certain that the value is 0.5". With the uniGauss model the entropy and the real value are separate. The variance encodes the uncertainty about the real value, and the bias can encode information about the uncertainty of using this particular Gaussian. The mean of the Gaussian encodes the expected real value. For the uniGauss to encode "I am very certain that the value is 0.5", it would put the mean at 0.5, and make the variance small.

However, despite the apparent benefits of a uniGauss model, I found that it was difficult to fit the model in practice. In particular, the binary units in the trained uniGauss models tended to either be always on or always off, which resulted in a simple Gaussian model. This might be because the free energy was being fit to the value of state-action pairs. If a state-action pair has a large negative value, then the way to represent this in the model is by placing all Gaussians far from that pair. But if a Gaussian is not near any state-action pairs, then it will always be turned off.

4.6 Learning Model Parameters

The model parameters are adjusted so that the goodness of a state-action pair under the product model approximates its action-value. I will use the modified temporal difference and Q-learning update rules SARSA(0) and Q(0). As we saw in chapter 2, the temporal-difference error quantifies the inconsistency between the value of a state-action pair and the discounted value of the next state-action pair, taking the immediate reinforcement into account. Values will be approximated by the negative free-energy of the PoE model:

$$\begin{aligned} Q(\mathbf{s},\mathbf{a}) &\approx -F(\mathbf{s},\mathbf{a}) \\ &= -\left\langle E(\mathbf{s},\mathbf{a},\mathbf{h})\right\rangle_{P(\mathbf{h}|\mathbf{s},\mathbf{a})} + H(P(\mathbf{h}|\mathbf{s},\mathbf{a})) \end{aligned} \tag{4.11}$$

To update the parameters of the model I use the SARSA temporal difference update rule or the Q-learning update rule.

4.6.1 SARSA

One possibility is to use a modified SARSA(0) learning rule designed to minimize the TD error [Rummery and Niranjan 1994; Sutton 1996]. Consider the case of the RBM . The SARSA update is a delta-rule update where the target for input $(\mathbf{s}^t, \mathbf{a}^t)$ is $r^t + \gamma Q(\mathbf{s}^{t+1}, \mathbf{a}^{t+1})$. The update for w_{ik} is given by:

$$\Delta\ w_{ik} \ \propto\ \left(r^t + \gamma Q(\mathbf{s}^{t+1}, \mathbf{a}^{t+1}) - Q(\mathbf{s}^t, \mathbf{a}^t)\right) \frac{\partial Q(\mathbf{s}^t, \mathbf{a}^t)}{\partial\ w_{ik}} \tag{4.12}$$

where for the restricted Boltzmann machine, the partial derivative with respect to a weight is give by:

$$\frac{\partial Q(\mathbf{s},\mathbf{a})}{\partial\ w_{ik}} \ = \ \frac{\partial}{\partial\ w_{ik}} \left[\sum_{k=1}^{K} \left(\sum_{i=1}^{|\mathcal{S}|} (w_{ik} s_i \widehat{h}_k + b_i s_i) + \sum_{j=1}^{|\mathcal{A}|} (u_{jk} a_j \widehat{h}_k + b_j a_j) \right) + \sum_{k=1}^{K} b_k \widehat{h}_k \right.$$

$$- \sum_{k=1}^{K} \widehat{h}_k \log \widehat{h}_k - (1 - \widehat{h}_k) \log (1 - \widehat{h}_k) \Bigg] \tag{4.13}$$

$$= s_i \widehat{h}_k \tag{4.14}$$

The other weights and biases are updated similarly:

$$\frac{\partial Q(\mathbf{s}, \mathbf{a})}{\partial \ u_{jk}} = a_j \widehat{h}_k \tag{4.15}$$

$$\frac{\partial Q(\mathbf{s}, \mathbf{a})}{\partial \ b_i} = s_i \tag{4.16}$$

$$\frac{\partial Q(\mathbf{s}, \mathbf{a})}{\partial \ b_j} = a_j \tag{4.17}$$

$$\frac{\partial Q(\mathbf{s}, \mathbf{a})}{\partial \ b_k} = \widehat{h}_k \tag{4.18}$$

For real-valued RBM-Rate networks, the parameter updates are identical. For uni-Gauss models, the means of the Gaussians are updated as above, with

$$\frac{\partial Q(\mathbf{s}, \mathbf{a})}{\partial \ \mu_{ik}} = \frac{\partial}{\partial \ \mu_{ik}} \Bigg[\sum_k \widehat{h}_k \left(b_k - \frac{1}{2} \log 2\pi - \frac{1}{2} \log v_k - \frac{\left\| [\mathbf{s} \ \ \mathbf{a}]^\top - \mu_k \right\|_2}{2 v_k^2} \right)$$

$$- \sum_k \Big(\widehat{h}_k \log(\widehat{h}_k) + (1 - \widehat{h}_k) \log(1 - \widehat{h}_k) \Big) \Bigg] \tag{4.19}$$

$$= -\widehat{h}_k (s_i - \mu_{ik}) / v_k^2 \tag{4.20}$$

The biases and log-variances are updated similarly:

$$\frac{\partial Q(\mathbf{s}, \mathbf{a})}{\partial \ b_k} = \widehat{h}_k \tag{4.21}$$

$$\frac{\partial Q(\mathbf{s}, \mathbf{a})}{\partial \ \log(v_k^2)} = -\widehat{h}_k \left[1 - (s_i - \mu_{ik})^2 / v_k^2 \right] \tag{4.22}$$

Although there is no proof of convergence in general for this learning rule, it works well in practice even though it ignores the effect of changes in parameters on $Q(\mathbf{s}^{t+1}, \mathbf{a}^{t+1})$. It is possible to derive update rules that use the actual gradient. See for example [Baird

and Moore 1999].

4.6.2 Q-learning

As discussed in chapter 2, SARSA is an on-policy update. That is, the algorithm is chasing a moving target, trying to converge to the value function of the policy currently being executed. The alternative is to use an off-policy method that always tries to converge to the value function of the optimal policy regardless of what policy is currently being executed. The Q(0) update rule is one such off-policy algorithm. I used the following update rule:

$$\Delta\ w_{ik}\ \propto\ \left(r^t + \gamma Q(\mathbf{s}^{t+1}, \mathbf{a}^*) - Q(\mathbf{s}^t, \mathbf{a}^t)\right) \frac{\partial Q(\mathbf{s}^t, \mathbf{a}^t)}{\partial\ w_{ik}} \tag{4.23}$$

where the partial derivative of the action-value function with respect to a parameter is the same as in Eq.(4.14). The only difference from SARSA(0) is the use of the action $\mathbf{a}^*$. This denotes an action sampled at a low temperature. Of course it is not guaranteed to be the optimal action, but it was typically a good action in the experiments that I tried (see section 4.8). Unlike SARSA(0), Q(0) is not guaranteed to converge to a bounded region of value function space, even in the linear case. In fact, examples of divergence are known [Bertsekas and Tsitsiklis 1996]. However, it works well in simulation in some cases.

4.7 Sampling Actions

Given a trained network and the current state $\mathbf{s}^t$, we need to generate actions according to their negative free energy. We do this with Gibbs sampling. There are (at least) two possible ways of implementing the Gibbs sampling, each with its advantages and drawbacks.

4.7.1 Sequential Action Sampling

We can iterate through the action units, sampling each in turn while holding the others fixed and integrating out the hidden units. In the discrete case we evaluate the free energy for each of an action variable's possible values while holding the other action units fixed. We then sample its value from a Boltzmann distribution. In other words, when updating the action units, we use a "softmax" to enforce the one-of-N constraint within a set of binary units that represent mutually exclusive actions of the same action variable. When the iterative Gibbs sampling reaches equilibrium it draws unbiased samples of action values according to their Q-value.

A nice feature of this method is that we avoid some sampling noise by explicitly integrating out the hidden units. This is possible because inference is tractable in the PoE model. One drawback is that it is less obvious how to parallelize sampling, although it is possible to do so (see [Neal 1999]). On serial computers any speed differences will be caused primarily by implementation details.

4.7.2 Alternating Block Sampling

Another method is alternating block sampling. This method will be appropriate when the undirected graphical model of the PoE has a bipartite graph structure. We start with an arbitrary initial action represented on the action units. Holding the state units fixed we update all of the hidden units in parallel so that we get a sample from the posterior distribution over the hidden units given the state and the action. Then we update all of the action units in parallel so that we get a sample from the posterior distribution over actions given the states of the hidden units. When updating the action units, we use a "softmax" to enforce the one-of-N constraint within a set of binary units that represent mutually exclusive settings of the same action variable. When the alternating Gibbs sampling reaches equilibrium it draws unbiased samples of hidden unit-action

pairs according to their value. We can then "marginalize" by throwing away the hidden activities and just keeping the action values.

A nice feature of this method is that the sampling of all hidden units can be conducted in parallel. Similarly (for the PoE models I consider) all action variables can be sampled in parallel. This is especially advantageous in large action spaces.

A problem with block sampling is that we frequently want to find actions sampled from a Boltzmann distribution at a temperature other than one. If we just alternately sample from the hidden and action variables at a lower temperature, we will not select actions according to the correct probabilities. We will be more likely to sample hidden-action pairs with low joint energy. These good hidden-action pairs may not correspond to good actions.

We can account for this by replicating the hidden units during sampling [Neal, personal communication]. Suppose we want to sample actions at temperature $T = 1/K$ for some positive integer K. We want a sample from a distribution proportional to $P(\mathbf{a}|\mathbf{s})^K$. Consider a network with K copies of the hidden units. The joint distribution of the actions and all replicas of the hidden units is given by:

$$\widetilde{P}(\mathbf{a}, \mathbf{h}_1, ..., \mathbf{h}_K|\mathbf{s}) \quad \propto \quad \exp\{-E(\mathbf{s}, \mathbf{a}, \mathbf{h}_1) - ... - E(\mathbf{s}, \mathbf{a}, \mathbf{h}_K)\}$$

with marginal:

$$\begin{aligned}
\widetilde{P}(\mathbf{a}|\mathbf{s}) &\propto \sum_{h_1,...,h_K} \exp\{-E(\mathbf{s}, \mathbf{a}, \mathbf{h}_1) - ... - E(\mathbf{s}, \mathbf{a}, \mathbf{h}_K)\} \\
&= \sum_{h_1} \exp\{-E(\mathbf{s}, \mathbf{a}, \mathbf{h}_1)\} \times ... \times \sum_{h_K} \exp\{-E(\mathbf{s}, \mathbf{a}, \mathbf{h}_K)\} \\
&\propto P(\mathbf{a}|\mathbf{s})^K \\
&= P(\mathbf{a}|\mathbf{s})^{1/T}
\end{aligned}$$

After sampling from the joint distribution of the larger network, we can get a sample

from the correct marginal by simply discarding the hidden unit values. This method has been proposed as a general optimization method by Doucet, Godsill, and Robert [2000].

4.7.3 Local Minima

Since we are explicitly sampling from the hidden units, the block sampling method is more sensitive to local minima in the joint hidden unit-action energy surface. I tested both kinds of sampling on 3500 random networks with two hidden and two action variables. Weights and biases were chosen from a Gaussian distribution with zero mean and a standard deviation of 10. Brief sampling lasted for 10 iterations. All sampling was done at a temperature of one. There were local minima problems with approximately two percent of the networks. The network shown in figure 4.4 is the worst example of the problem that I found.[2]

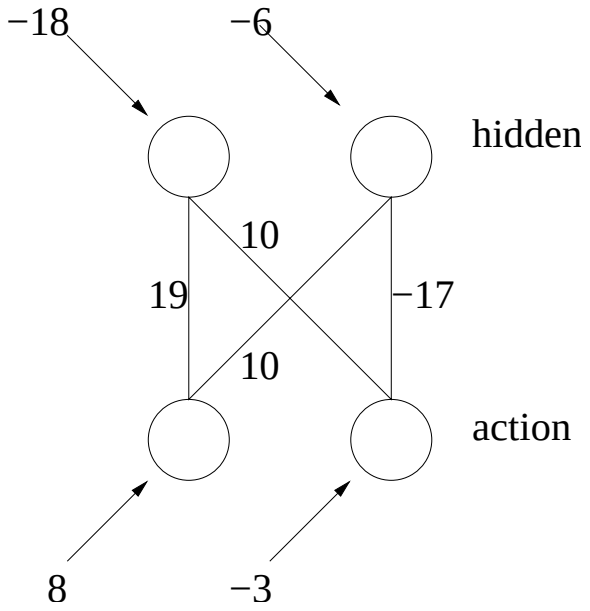

Figure 4.4: An example network that is troublesome for block sampling.

In this case the best action is [1 1] (F=-16). The next best action is [1 0] (F=-13.3). The conditional distributions over hidden states and actions are shown in figure 4.5.

Iteratively sampling between hidden units and actions causes the sampler to often get

[2]That is, the example for which the frequencies over actions given by the two sampling methods were most different in the sense of squared difference.

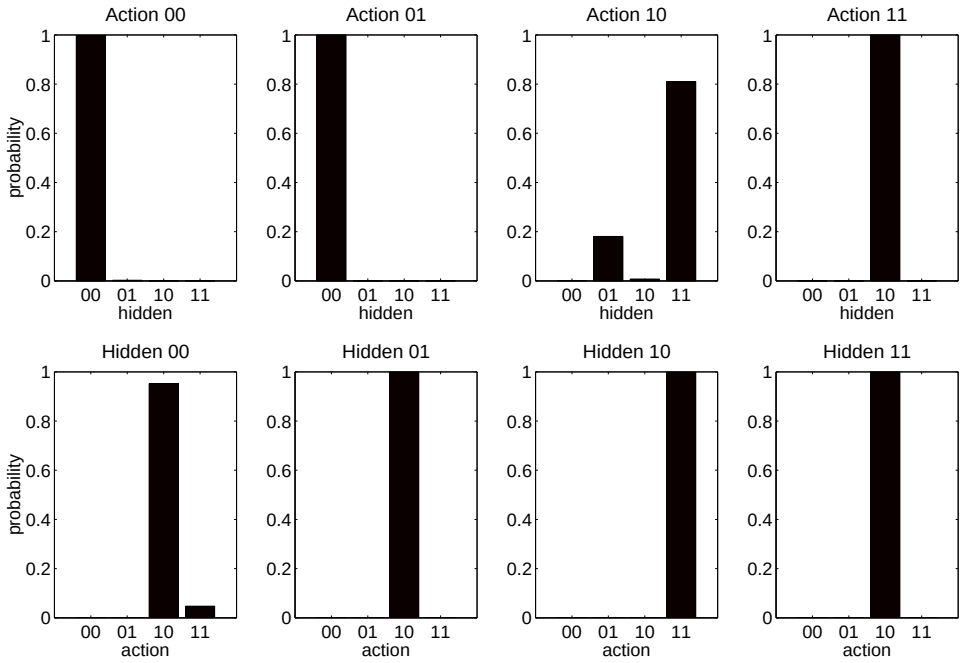

Figure 4.5: Conditional distributions used by the block sampler for the problematic example. The top four plots show the conditional distribution over each hidden configuration given the action configuration. The bottom four plots show the conditional distribution over each action configuration given the hidden configuration. At this temperature we will almost always get stuck in the suboptimal action. Using sequential sampling we can avoid this local minima.
.

stuck with the suboptimal (hidden,action) configuration ([1 1],[1 0]). Given an initialization of the action to [0 0], we would expect to almost always end up at the suboptimal action. We would expect to jump out of the suboptimal minima with probability of around 0.008 for each iteration of sampling at temperature $T = 1$. Given enough time, we would expect the sampler to find the better action. However, because we are using brief sampling, there is not enough time to find the good action in this case. Figure 4.6 shows the frequency of ending up in each of the four energy states using the two types of sampling, given an initial action of [0 0].

In the 3500 test cases, I did not find an example of a network where block sampling worked better than sequential sampling. Of course there should be some types of networks for which one or the other type of sampling is more prone to getting stuck in local minima.

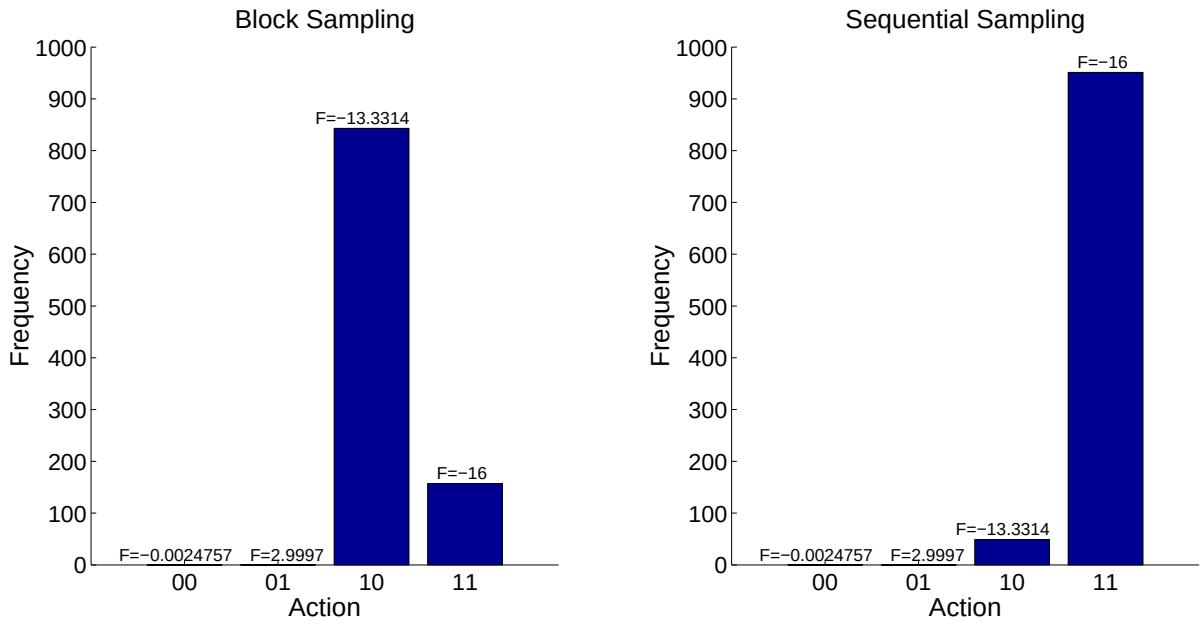

Figure 4.6: Frequency versus action number for the two types of Gibbs sampling. The free energy of the configuration is shown above the bar. In this case the block sampling method frequently gets stuck in a local minimum.

In general, block sampling should be more sensitive to local minima involving hidden state configurations, while sequential sampling should be more sensitive to local minima for which you have to change several action variables simultaneously to escape. In practice I found that using sequential sampling or block sampling resulted in similar performance on the various experimental tasks. Because the experiments were implemented in Matlab, block sampling was faster in my case. I was able to make use of the fast Matlab linear algebra routines to quickly sample from all hidden units or all actions units in a few Matlab operations. In contrast, sequential action sampling involves iterating over action variables. This type of iteration is slow in Matlab. In another language (such as C) this would not be such an issue.

4.7.4 Exploration

Given that we can select actions according to their value, we still have to decide on an exploration strategy. One common action selection scheme is Boltzmann exploration. The

probability of selecting an action is proportional to $e^{Q(s,a)/T}$, where Q is the action-value function. It can move from exploration to exploitation by adjusting the "temperature" parameter T. Another possible selection scheme is ϵ-greedy, where the optimal action is selected with probability $1 - \epsilon$ and a random action is selected with probability ϵ. The exploration probability ϵ can be reduced over time, to move the learner from exploration to exploitation.

If the SARSA(0) update rule is used with Boltzmann exploration then samples from the Boltzmann distribution at the current temperature are sufficient. However, if Q(0) or ϵ-greedy is used we must also evaluate $\max_a Q(s, a)$. This can be approximated by sampling at a low temperature. To improve mixing the temperature can be initialized to a high value and lowered during the course of sampling.

4.8 Simulation Results

To test the algorithm I introduce four tasks: The stochastic policy task; The blockers task; the large-action task; and the rocket task. Each task highlights a different aspect of the algorithm, its strengths and weaknesses.

4.8.1 The Stochastic Policy Task

This is an implementation of the example task mentioned in section 4.4.1. There is no (visible) state, and the action consists of N binary variables. Two settings of the action variables are randomly selected to be rewarded. In other words, there are two "good" actions that will be rewarded, and the rest of the possible 2^N actions will receive no reward.

Although it is not visible to the learner, there is a hidden state variable which indicates which of the two "good" actions is currently being rewarded. Executing the other "good" action will result in no reward. Executing the correct action results in a reward of +10,

and a toggling of the binary hidden variable. During the task, repeating the same "good" action multiple times will result in only a single reward after the first execution. Given access to the hidden state, the optimal policy would be to alternate the two "good" actions. Because the learner does not have access to the hidden state and has no memory the optimal policy is stochastic: Pick from the two "good" actions with equal probability.

A restricted Boltzmann machine with one hidden unit was trained on an instance of the task for $N = 5$. The Q-learning update rule was used, and training lasted for 20 000 trials. The result is shown in figure 4.7.

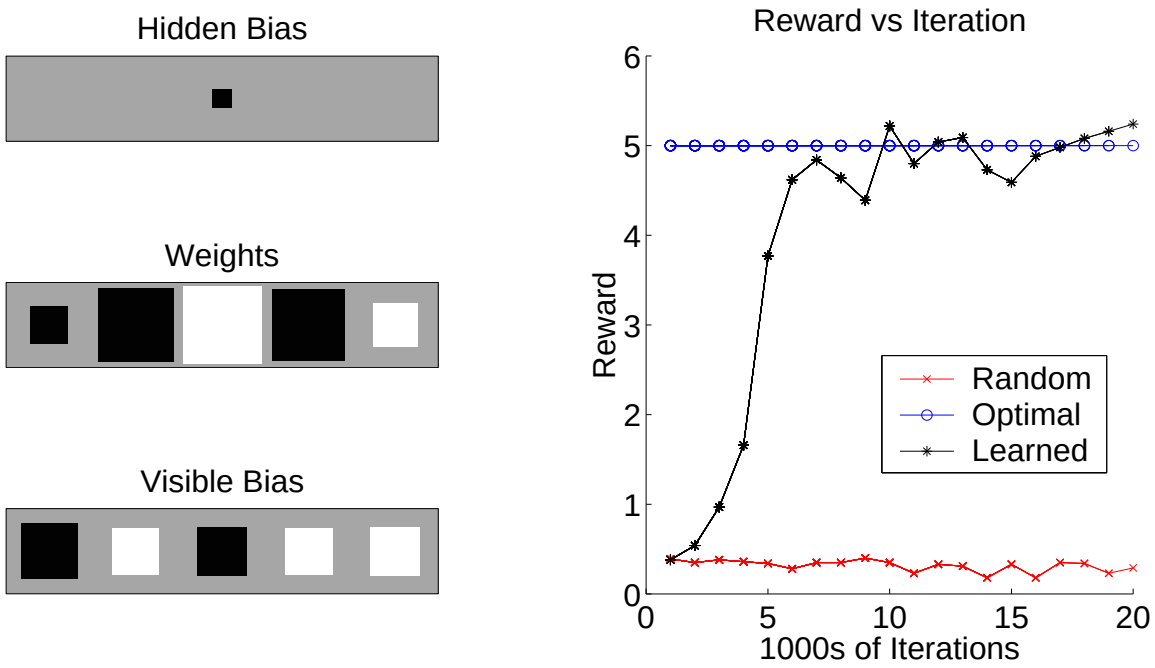

Figure 4.7: Results of learning an implicit stochastic policy. The Hinton diagram shows the sign and magnitude of the biases on units, and the weights from hidden to action units. White is positive and black is negative. The area of a square indicates the magnitude of the weight or bias.

The two "good" actions were {0 0 1 0 1} and {0 1 0 1 1}. The first is activated when the hidden unit is on, and the second is activated by the visible biases when the hidden unit is off. The hidden unit turns on with probability 0.4845. The graph shows a plot of reward vs. iteration of learning. The reward was found by executing the policy recommended by the current action-value function for 250 steps and averaging the

result. The "optimal" reward is the expected reward for picking randomly from the two rewarding actions with equal probability.

Not all of the action bits must alternate at each time step. Some of the action bits alternate, and others are held fixed. The learner must discover what subset of action variables must alternate. In the example, the agent has learned to always switch off the first action bit, and switch on the fifth. The others are anti-correlated via the hidden unit. A pre-specified family of stochastic policies would be able to achieve the same performance if it allowed for at least three action variables to be correlated (or anti-correlated). As the number of action variables grows, pre-specifying which correlations are allowed becomes prohibitive in terms of the number of parameters required.

4.8.2 The Blockers Task

The blockers task is a co-operative multi-agent task in which there are offensive players trying to reach an end zone, and defensive players trying to block them (see Figure 4.8).

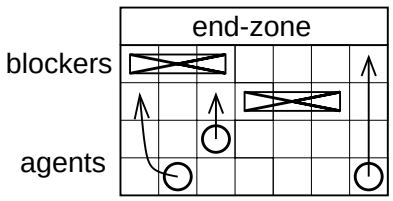

Figure 4.8: An example of the "blocker" task. Agents must get past the blockers to the end-zone. The blockers are pre-programmed with a strategy to stop them, but if the agents co-operate the blockers cannot stop them all simultaneously.

The task is co-operative: As long as one agent reaches the end-zone, the "team" is rewarded. The team receives a reward of $+1$ when an agent reaches the end-zone, and a reward of -1 otherwise. The blockers are pre-programmed with a fixed blocking strategy. Each agent occupies one square on the grid, and each blocker occupies three horizontally adjacent squares. An agent cannot move into a square occupied by a blocker or another agent. The task has non-wrap-around edge conditions on the bottom, left and right sides

of the field, and the blockers and agents can move up, down, left or right. Agents can not move in to squares occupied by other agents or by blockers. Agents are ordered. If two agents want to move in to the same square, the first agent in the ordering will succeed, and any others trying to move into that square will be unsuccessful. Note that later agents can not see the moves of earlier agents when making decisions. The ordering is just used to resolve collisions. If a move is unsuccessful, then the agent remains in its current square.

The blockers moves are also ordered, but subsequent blockers do make decisions based on the moves of earlier blockers. The blockers operate a zone-based defense. Each blocker takes responsibility for a group of four or five columns. For example, blocker 1 is responsible for columns 1 through 4, blocker 2 is responsible for columns 4 through 8, and so on. If an agent moves into one of its columns and is in front of the end-zone, a blocker will move to block it. Because of the ordering, blockers will not move to stop agents that have already been stopped by other blockers.

A restricted Boltzmann machine with 4 hidden units was trained using the Q-learning rule on a 5×4 blocker task with two agents and one blocker. The combined state consisted of three position variables (two agents and one blocker) which could take on integer values $\{1, ..., 20\}$. The combined action consisted of two action variables taking on values from $\{1, ..., 4\}$.

The network was run twice, once for 60 000 combined actions and once for 400 000 combined actions, with a learning rate going from 0.1 to 0.01 linearly and temperature going from 1.0 to 0.01 exponentially over the course of training.[3] Each trial was terminated after either the end-zone was reached, or 20 combined actions were taken, whichever occurred first. Each trial was initialized with the blocker placed randomly in the top row and the agents placed randomly in the bottom row. The same learning rate and temperature schedule were used to train a Q-learner with a table containing 128 000

[3]These learning parameters were selected by trial and error during preliminary experiments.

elements ($20^3 \times 4^2$), except that the Q-learner was allowed to train for 1 million combined actions. After training each policy was run for 10 000 steps, and all rewards were totaled. The two algorithms were also compared to a hand-coded policy, where the agents first move to opposite sides of the field and then move to the end-zone. In this case, all of the algorithms performed comparably, and the PoE network performs well even with very few training iterations (see Table 4.2). The variation in results between the hand-coded policy and the PoE was caused by the random starting positions of agents and blockers.

An RBM network with 16 hidden units was trained on a 4×7 blockers task with three agents and two blockers. Again, the input consisted of position variables for each blocker and agent, and action variables for each agent. The network was trained for 400 000 combined actions, with the a learning rate from 0.01 to 0.001 and the same temperature schedule as the previous task. Each trial was terminated after either the end-zone was reached, or 40 steps were taken, whichever occurred first. After training, the resultant policy was run for 10 000 steps and the rewards received were totaled. As the table representation would have over a billion elements ($28^5 \times 4^3$), a table based Q-learner could not be trained for comparison. The hand-coded policy moved agents to the left, middle and right column, and then moved all agents toward the end-zone. The PoE performed slightly worse than this hand-coded policy. The results for all experiments are summarized in Table 4.2. Note that all reward totals were negative, so a smaller magnitude is better.

An example of a typical run for the 4×7 task is shown in figure 4.9. The strategy discovered by the learner is to force the blockers apart with two agents, and move up the middle with the third. Examples of features learned by the experts are shown in figure 4.10. The hidden units become active for a specific configuration in state space, and recommends a specific set of actions. Histograms below each feature indicate when that feature tends to be active during a trial. The histograms show that feature activity is localized in time. Features can be thought of as macro-actions or short-term

Table 4.2: PoE "Blocker" Experimental Results

Algorithm	Reward
Random policy (5×4, 2 agents, 1 blocker)	-9986
hand-coded (5×4, 2 agents, 1 blocker)	-6782
Q-learning (5×4, 2 agents, 1 blocker, 1000K steps)	-6904
PoE (5×4, 2 agents, 1 blocker, 60K steps)	-7303
PoE (5×4, 2 agents, 1 blocker, 400K steps)	-6738
Random policy (4×7, 3 agents, 2 blockers)	-9486
hand-coded (4×7, 3 agents, 2 blockers)	-7074
PoE (4×7, 3 agents, 2 blockers, 400K steps)	-7631

policy segments. Each hidden unit becomes active during a particular "phase" of a trial, recommends the actions appropriate to that phase, and then ceases to be active.

4.8.3 The Large-Action Task

The solution to the blockers task shows that free energy can be used to approximate an action-value function. While the task is made more difficult by prolonged temporal delays in reward, the blockers task does not have a very large action space. The following task has the opposite properties: It has a large action space, but has no temporal delays in reward.

Consider a version of the task with an N-bit action. The task is generated as follows: Some small number of state-action pairs are randomly selected. I will call these "key" pairs. During execution, a state is chosen at random, and an action is selected by the learner. A reward is then generated by finding the key state closest to the current state (in Hamming distance). The reward received by the learner is equal to the number of bits that match between the action for this key state and the current action. So if the agent selects the key action it receives the maximum reward of N. The reward for any other action is found by subtracting the number of incorrect bits from N. The task (for

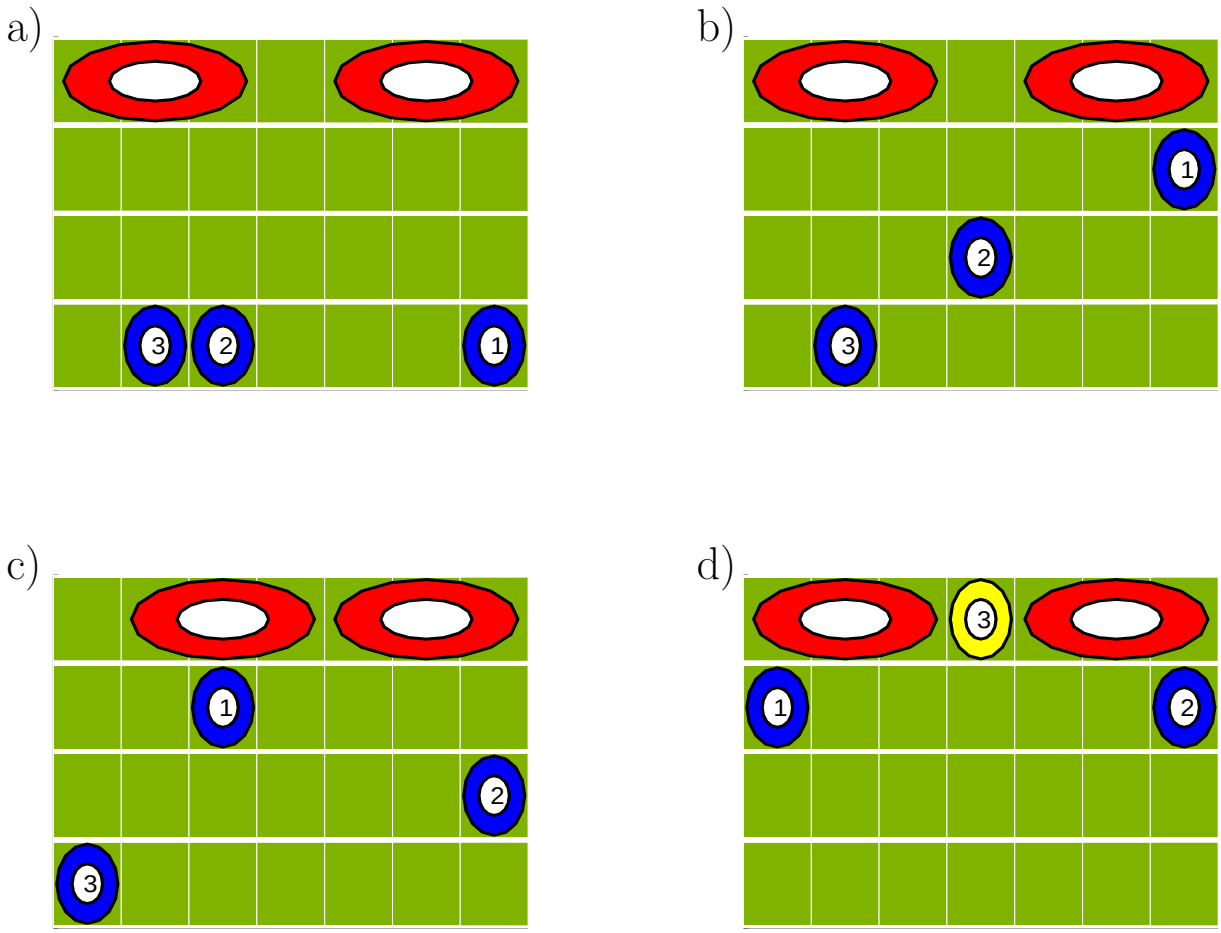

Figure 4.9: Example agent strategy after learning the 4×7 blocker task. a) The three agents are initialized to random locations along the bottom of the field. b) Two of the agents run to the top of the playing field. c) These two agents split and run to the sides. d) The third agent moves up the middle to the end-zone.

$N = 5$) is illustrated in figure 4.11.

A PoE network with nine hidden units was trained on an instantiation of the large action task with an eight-bit state space and a 40-bit action space. Nine key states were randomly selected. The network was run for 12 000 actions with a learning rate going from 0.1 to 0.01 and temperature going from 1.0 to 0.1 exponentially over the course of training.[4] Each iteration was initialized with a random state. Each action selection consisted of 200 iterations of Gibbs sampling.

Because the optimal action is known for each state we can compare the results to the

[4]These learning parameters were selected by trial and error during preliminary experiments.

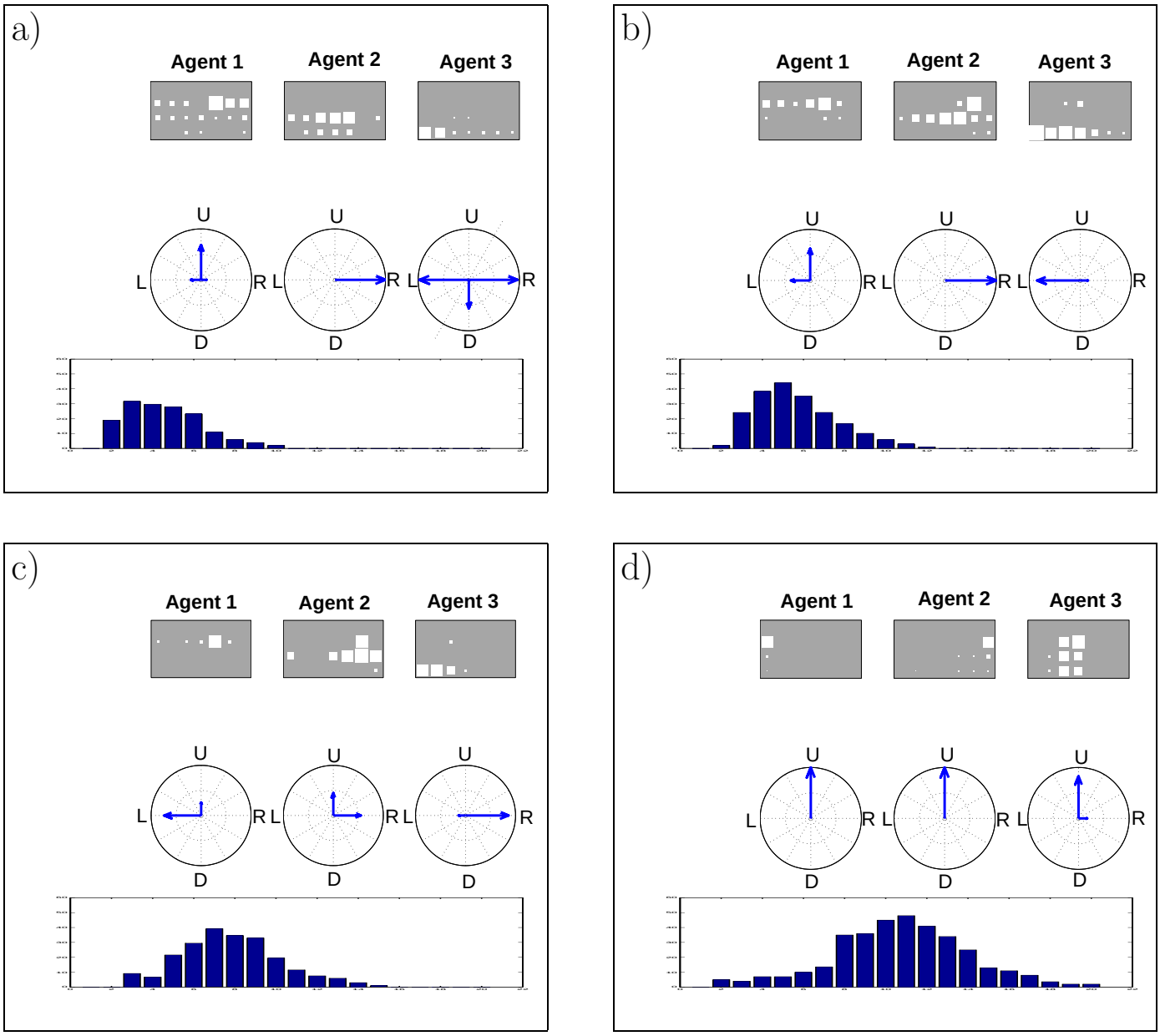

Figure 4.10: Features of the learned value function approximator. The four features (a,b,c and d) correspond to the four stages shown in figure 4.9. Each feature corresponds to a hidden unit in the RBM. The Hinton diagram shows where each of the three agents must be in order to activate the feature. The vector diagram indicates what actions are recommended by the feature. The histogram is a plot of frequency of activation of the feature versus time in a trial. It shows when during a run this feature tends to be active. The learned features are localized in state space and action space. Feature activity is localized in time.

optimal policy. Because of the size of the state and action space, I do not compare to a table-based learner or a hand-crafted policy. The results are shown in figure 4.12.

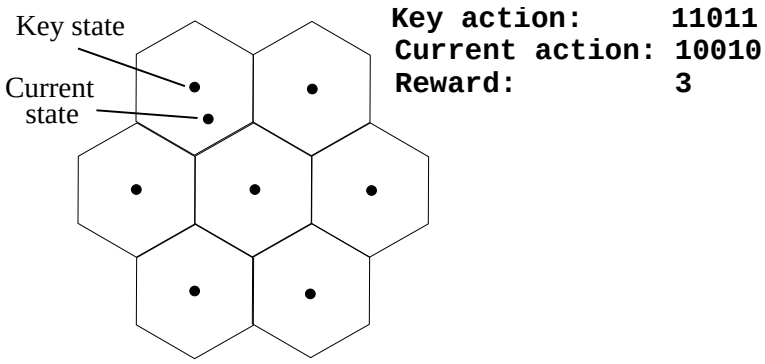

Figure 4.11: The large action task. The space of state bit vectors is divided into clusters of those which are nearest to each "key" state. Each key state is associated with a key action. The reward received by the learner is the number of bits shared by the selected action and the key action for the current state.

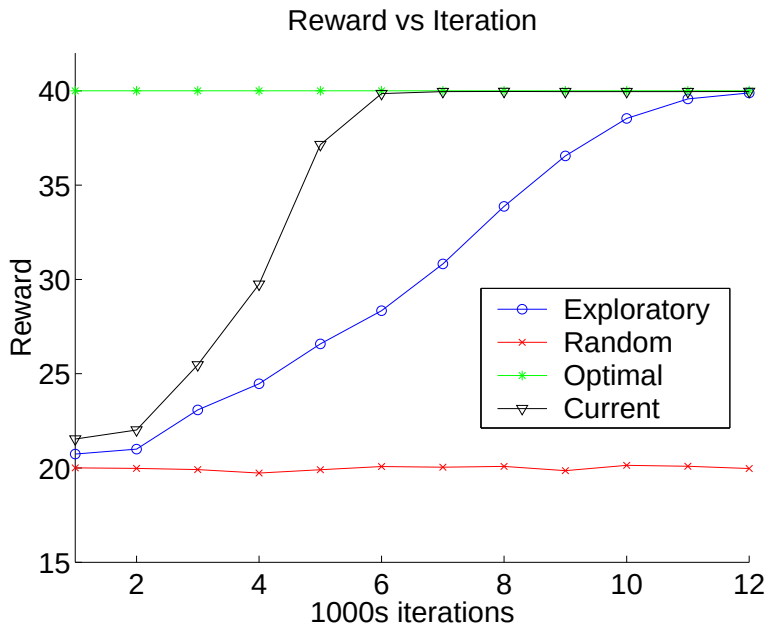

Figure 4.12: Results for the large action task. The optimal policy gives an average reward of 40. "Random" shows the average return for a random action selection policy. "Exploratory" shows the reward returned by the current value function at the current exploration temperature. "Current" shows the reward returned by the current value function at the minimum exploration temperature. This last curve shows the greedy policy with respect to the current value function.

The learner must overcome two difficulties. First, it must find actions that receive rewards for a given state. Then, it must cluster the states which share commonly rewarded actions to infer the underlying key states. As the state space contains 2^8 entries and

the action space contains 2^{40} entries, this is not a trivial task. Yet the learner achieves perfect performance after 12 000 actions.

4.8.4 The Rocket Task

This task involves factored, real-valued state and action variables. A rocket must be guided to a target. The state consists of six real-valued variables: x-y position; x-y velocity; and x-y target position. The position variables are in the range $[0, 1]$, and the velocity variables are in the range $[-10, 10]$. The action consists of two real-valued variables: x and y thrust. The action variables are in the range $[-10, 10]$. Immediate reward is based on current distance to target, with a bonus if the target is reached:

$$r_t = (\sqrt{2} - \sqrt{(x - t_x)^2 + (y - t_y)^2}) + 5\mathrm{h}(t) \tag{4.24}$$

where $\mathrm{h}(t)$ is an indicator function which is unity if the target is reached during time step t, and zero otherwise.

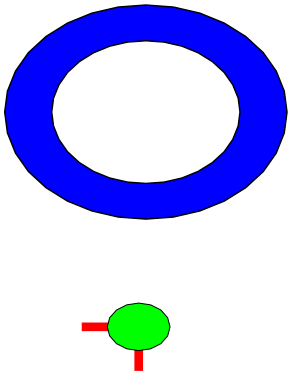

Figure 4.13: The rocket task. The rocket (green circle) must reach the target (blue ring). The state consists of x-y position and velocity, and target x-y position. The action is the x-y thrust (red lines).

A rate-coded restricted Boltzmann machine with 10 hidden units was trained on the rocket task. Each trial of the task lasted for 200 steps, or until the target was reached, whichever came first. The network was trained for 10 000 combined actions using the SARSA update rule with a learning rate going from 0.1 to 0.01 linearly and temperature

going from 5.0 to 0.1 exponentially over the course of training. Each trial was initialized with the rocket in a random position, with random velocity, and with the target in a random position.

After training the policy was run for 10 000 trials, and all rewards were totaled. The result of training was compared to a random policy and the optimal policy (which is to head straight for the target at maximum thrust). The results are shown in figure 4.8.4. We can see that the learned policy does nowhere near as well as the optimal policy.

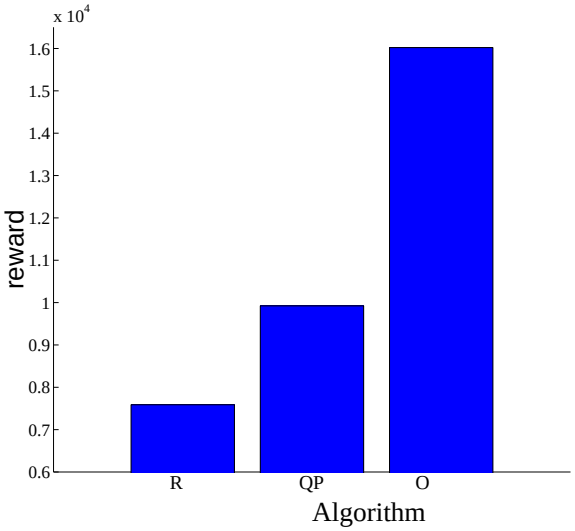

Figure 4.14: Results on the rocket task. R = Random policy; QP = Q-learning with PoE; O = Optimal

This is due to the limitations of the RBMrate network in coding real-valued variables. Variables tended to saturate at on or off, leading to "bang-bang" type control along the x and y axes. This can be seen in figure 4.15.

A tractable mixture of Gaussians model was also trained on the rocket task. The mixture contained 128 elements. When the target position was held fixed at the center, performance was comparable to the RBMrate network. However, when the target location was allowed to vary, the performance degraded completely. The mixture model was unable to deal with the larger state space. A uniGauss network was also trained on the rocket task. Although the uniGauss network has better representational properties, I found that it behaved poorly in practice. In fact, I was unable to successfully fit a uniGauss model to this task. The result was always the same: The binary variables were

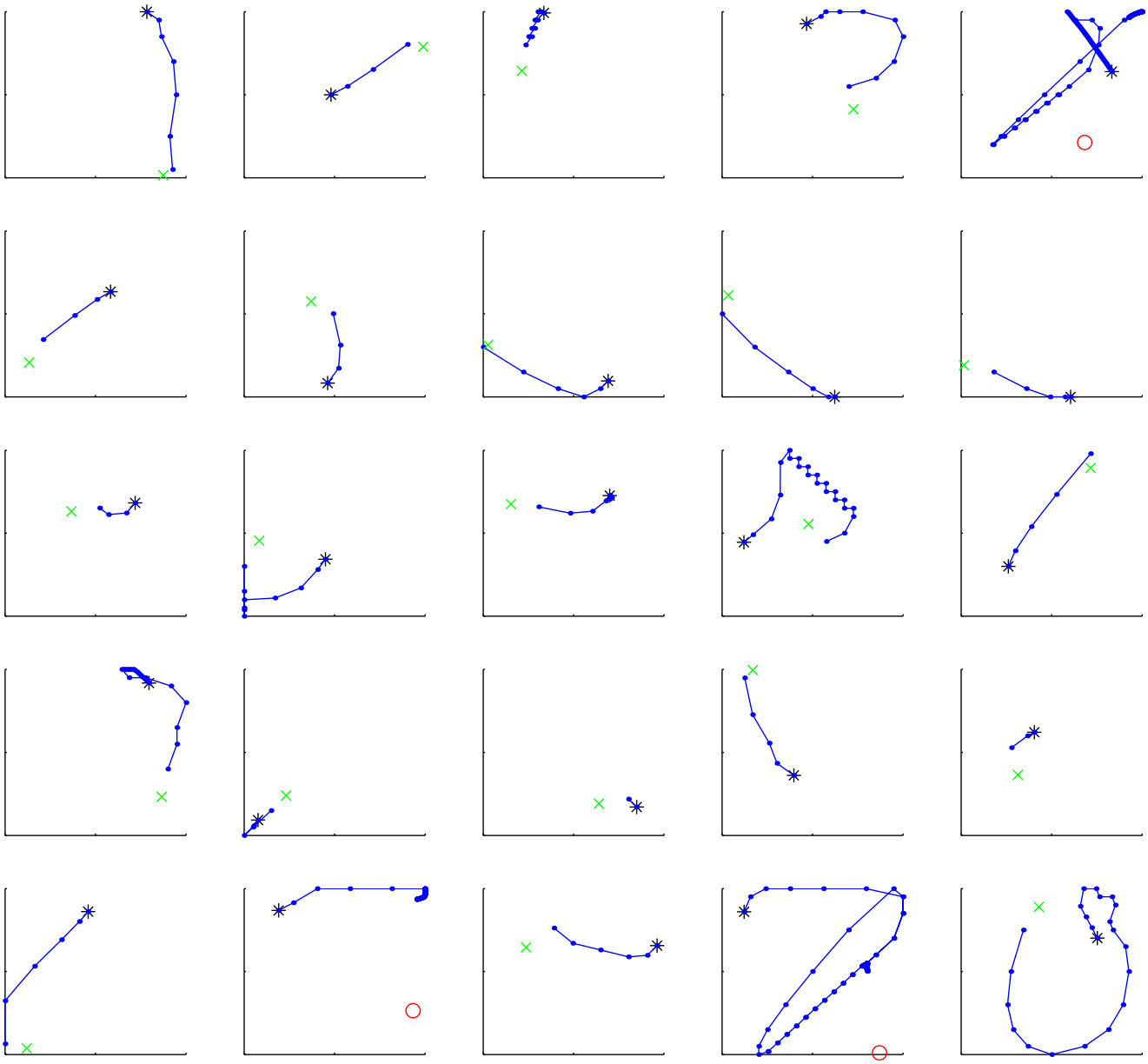

Figure 4.15: Example paths for RBMrate after training. The asterisk "$*$" indicates the rocket starting position. The other mark indicates the target position. "X" shows that the target was hit before the trial finished. "O" indicates a miss. Dots along the trajectory indicate equally spaced points in time. Typically the control variables saturate, leading to "bang-bang" control.

biased to be either always on or always off. It is possible that new models in the PoE family which are better at representing real values, such as Frequently Approximately Satisfied models [Hinton and Teh 2001], will help improve performance on tasks with

real-valued states and actions.

Chapter 5

Discussion

5.1 States

The DBN algorithm is related to various streams of research in machine learning and reinforcement learning. Factored dynamic models have been learned from data by Ghahramani and Jordan [1997]. They used a structured variational approximation and batch updating. Non-factored models have been learned for POMDPs [Chrisman 1992; McCallum 1995]. They used split/merge based model selection techniques to determine the size of the required state space. The split/merge criteria were designed either to maximize likelihood [Chrisman 1992] or maximize the ability of the model to predict values [McCallum 1995]. Using pre-specified factored models for dynamic programming [Boutilier and Poole 1996; Meuleau, Hauskrecht, Kim, Peshkin, Kaelbling, Dean, and Boutilier 1998; Dean, Givan, and Kim 1998; Poupart and Boutilier 2000; Boutilier, Dearden, and Goldszmidt 2000; Charnes and Shenoy 1999; **?**; Poupart, Ortiz, and Boutilier 2001] and reinforcement learning [McCallum 1995; Rodriguez, Parr, and Koller 2000; Thrun 2000] has also been investigated. Some of these have used MCMC approximations [Thrun 2000; Poupart, Ortiz, and Boutilier 2001] or belief state simplification [Rodriguez, Parr, and Koller 2000; Poupart and Boutilier 2000] to approximate inference in a POMDP.

Factored models of fully-observable MDPs were learned in [Tadepalli and Ok 1996]. To my knowledge this work is the first attempt to combine learning of a factored model, using approximate inference and reinforcement learning.

5.1.1 Model Features

I have shown that learning and using a compact representation of the core state of a POMDP is feasible. The DBN model performs as well as other methods on problems with a smaller state space, and outperforms localist models on more complex problems with high-dimensional inputs and substantial hidden state.

The units in the DBN encode useful features of the input, such as whether a car was at the "near" or "nose" position. They also encode dynamic history, such as current gaze direction or previous action. This has advantages over a simple stochastic policy learned via Q-learning: If the Q-learner knows that there is an oncoming car, it can at best randomly select to look left or right. The DSBN systematically looks to the left, and then to the right, wasting fewer actions.

It is interesting to ask what information the units are encoding in the visual NY driving task. Since we have access to the factored visible state from the original NY driving task (table 3.1) we can compare hidden unit activities to the presence of these original attributes to get some insight into what the hidden units are encoding. I show the correlation between original state variables and hidden unit activity for some interesting units (figure 5.1). In particular, there are hidden units that encode world state (object type, object speed, object distance), and "proprioceptive" information (gaze direction). In the graphs, a unit was counted as "active" if the (approximated) posterior probability of turning the unit on was greater than or equal to 0.95. Remember, the model itself did not have access to the original sensory information. It just had access to the 8×8 image. It learned to model these attributes because they were useful for describing the dynamics or emissions of the process.

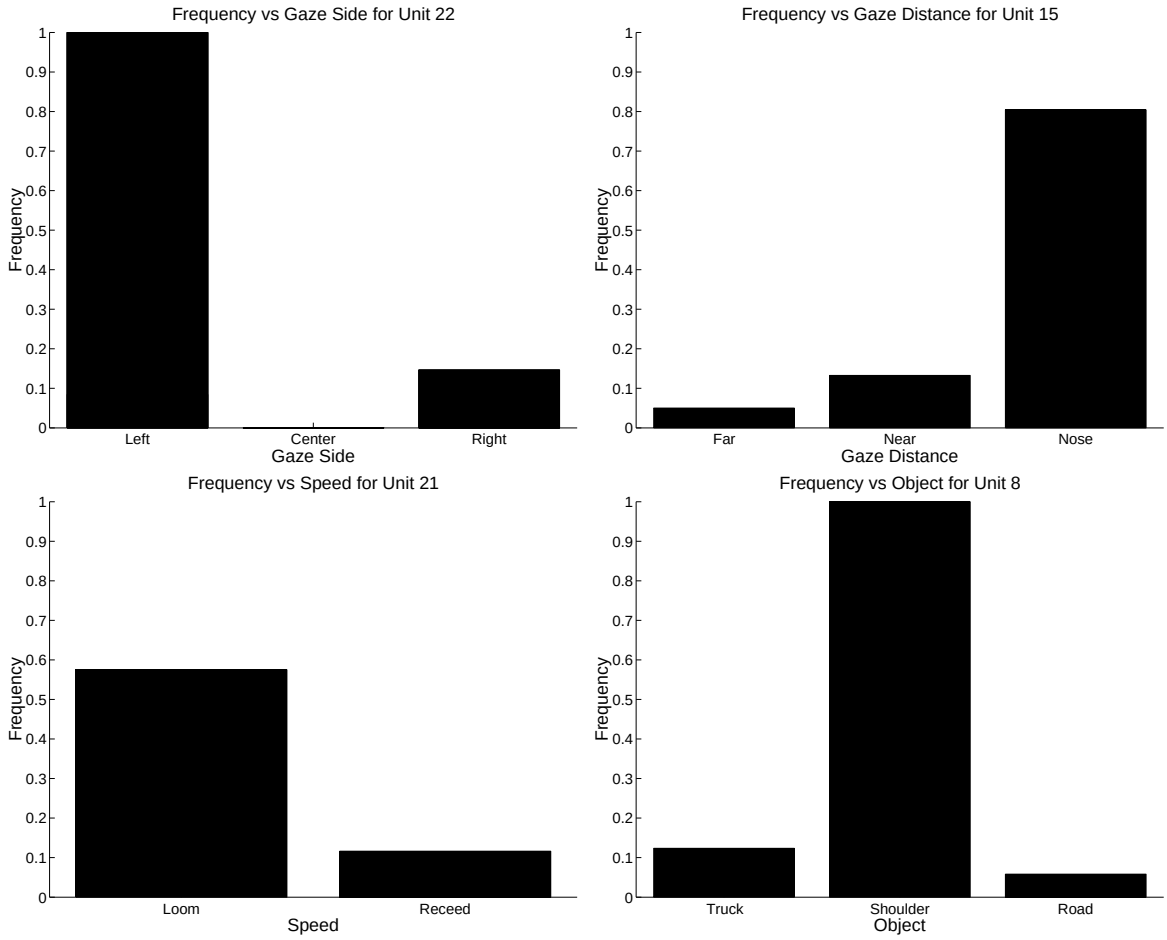

Figure 5.1: Correlation between state attributes and hidden unit activity for some hidden units. The y-axis indicates the proportion of the time that a hidden unit is active when the state attribute is present. The unit is acting as a detector for an attribute value when the proportion is near 1 for that value and near 0 for the others.

The DBN model is learned without regard to the task being solved. The localist models are also trained in an unsupervised way, but do not perform as well. This suggests that the DBN model is simply able to capture more of the dynamical structure than the localist models. Some of this structure turns out to be useful in predicting reward, which results in higher performance. It would be interesting to learn models that only have features that are useful in predicting reward. I discuss the possibility of learning task-specific models further in section 5.4.3.

5.1.2 Models of Reward

The DBN method requires a model in order to update the belief state at each step. Once a model is known we could do full iterations of dynamic programming instead of the sample backups of Q-learning. It is possible to do full backups efficiently when we have access to a compact representation of the process [Boutilier and Poole 1996]. This might speed convergence and improve the stability of the algorithm. Similarly, we could consider "batch" updates of the dynamical model, where the model is held fixed and experience is captured over a period of time. Then the model is updated to reflect this batch of experience.

Full dynamic programming backups also require a model of immediate reward. As described, the method did not include a model of immediate reward generation. We can learn a simple model by assuming that each core state variable contributes to the immediate reward independently:

$$r^t = \sum_{k=1}^{K} r_k^t s_k^t \tag{5.1}$$

where $r_k^t \sim \mathcal{N}(m_i, \psi_i)$ is drawn from a Gaussian with a learned mean and variance. The parameters of the Gaussian can be learned by stochastic gradient descent on the squared difference between the predicted and sampled immediate reward. Learning a model of instantaneous reward would be necessary for dynamic programming. It would also be useful for belief updating: The observed reward is evidence about our current state, and really should be used to update the belief state at each time step. This has not been done in the current method.

5.1.3 Convergence

A limitation of the DBN algorithm is in convergence. There are no formal guarantees that the learning rule for the value function will converge to the best approximation, or even

that the parameters will remain bounded. Since I am training the dynamic model using stochastic gradient descent, we are guaranteed that in the limit the DBN will converge to a local optimum in approximate log-probability.

Simulations have shown that convergence can be a problem for the DBN model. Some runs converge to a poorly-performing solution. Interestingly, the failures seem to be caused not by the value function, but by the model of the dynamics. Given a "good" model of the dynamics, the method was observed to always converge to a well-performing solution. This suggested that some regularization of the model would help get more consistent results. This was the reason I introduced ARD, and I found that ARD does improve convergence to good solutions. It might be that forcing the method to learn a more task-specific model would help even more. However, there would still be no formal guarantee of convergence for the value function. Using Monte Carlo methods to get unbiased estimates of expected return would allow us to learn an approximate value function with guaranteed convergence (subject to the usual restrictions on learning rates for the convergence of stochastic gradient ascent). However, using a full Monte Carlo method might require so much training time that the algorithm would no longer be practical. Also, local minima in the value function would still be a potential problem.

5.2 Actions

The action sampling method is closely related to actor-critic methods [Barto, Sutton, and Anderson 1983; Sutton 1990]. Normally with an actor-critic method, the actor network can be viewed as a biased scheme for selecting actions according to the value assigned by the critic. The selection is biased by the choice of parameterization. The sampling method of action selection is unbiased (if the Markov chain is allowed to converge). Further, the resultant policy can potentially be much more complicated than a typical parameterized actor network would allow. This is because a parameterized policy would

have to be explicitly normalized. This is exactly the trade-off explored in the graphical models literature between the use of Monte Carlo inference [Neal 1992] and variational approximations [Jaakkola 1997].

The sampling algorithm is also related to probability matching [Sabes and Jordan 1996], in which good actions are made more probable under a model, and the temperature at which the probability is computed is slowly reduced over time in order to move from exploration to exploitation and avoid local minima. Unlike the sampling algorithm, the probability matching algorithm used a parameterized distribution which was maximized using gradient descent, and it did not address temporal credit assignment.

5.2.1 Macro Actions

One way to interpret the individual models in the product are that they are learning "macro" or "basis" actions. As we have seen with the Blockers task, the hidden units come to represent sets of actions that are spatially and temporally localized. We can think of the hidden units as representing "basis" actions that can be combined to form a wide array of possible actions. The benefit of having basis actions is that it reduces the number of possible actions, thus making exploration more efficient. The drawback is that if the set of basis actions do not span the space of all possible actions, some actions become impossible to execute. By optimizing the set of basis actions during reinforcement learning, we find a set that can form useful actions, while excluding action combinations that are either not seen or not useful. The advantage of having a subset of basis actions can be seen by using a pre-learned action basis set, and comparing learning time to a model that must learn both state and action associations. I re-ran the 2-agent "blockers" task (see section 4.8.2) under four different conditions (see figure 5.2).

The first condition, "learning macros" is the same as the original experiment. All state-to-hidden and action-to-hidden weights are randomly initialized and learned simultaneously. The second condition, "optimized action macros", used a fixed set of

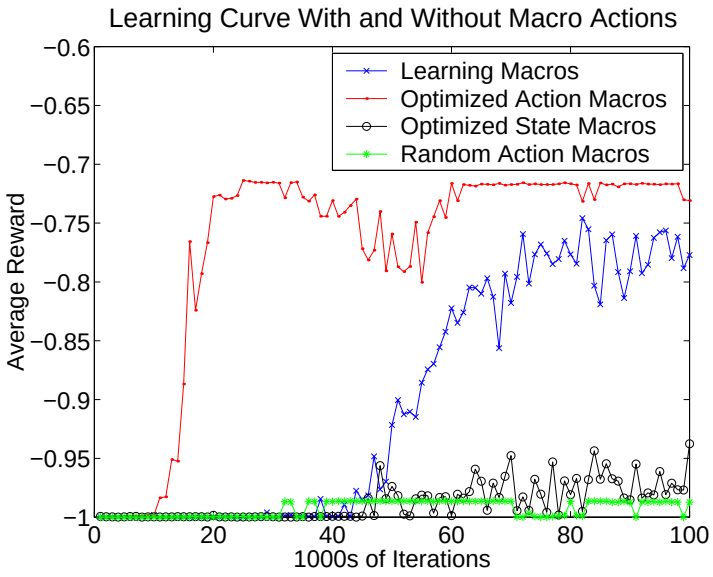

Figure 5.2: Learning curve for 2-agent "blockers" problem with and without pre-learned macro actions. The plot shows the average reward per time step for the current policy, averaged over 20 runs, versus training time in 1000s of time steps.

pre-learned action-to-hidden weights, and learns the state-to-hidden weights. The fixed action-to-hidden weights were taken from an earlier successful run of the algorithm. Similarly, the third condition, "optimized state macros", used a fixed set of optimized state-to-hidden weights and learns the action-to-hidden weights. The final condition used a fixed, random set of action-to-hidden weights, and optimizes the state-to-hidden weights.

Notice that the increased learning speed is not simply due to reducing the number of learned parameters. When the state-to-hidden associations are fixed in advance the model fails to learn. This asymmetry is related to the basic asymmetry of MDPs: The learner observes what state it is in, but must choose its action. Pre-selecting which combinations of states are represented by the hidden units does not make the task of exploration any easier.[1] The learner will still see the same combinations of states, and must still select from an exponential number of actions. However, it is not enough to simply fix some

[1]In fact it can make learning much harder if the states that can be represented are not the ones seen during early learning. This is probably the reason why the "optimized state macros" condition fails to learn. The states that can be represented in the value function are restricted to those seen during successful trials, not those seen at the beginning of learning and exploration. During early exploration many states are seen that would essentially all look the same to the value function approximator.

random macro-actions *a priori*. In the example, the number of hidden unit combinations is actually the same as the number of possible actions. Simply fixing random macro actions does not speed learning. When action macros are learned, however, they exclude some bad actions. For example, none of the learned macro actions make the agents move down. It is this restriction to a subset of good actions that speeds learning.

The "macro" actions learned by the PoE should not be confused with temporally abstract actions. The learning and use of temporally abstract actions is an important area of current research in reinforcement learning [Parr and Russell 1998; Precup, Sutton, and Singh 1998; McGovern 1998; Dietterich 2000]. The "macro" actions learned by the PoE have some features in common with these temporally abstract actions. In particular, the PoE macro actions tend to remain active for temporally prolonged periods. However, that does not make them temporally abstract actions. They do not come with the formal machinery of most temporally abstract actions (such as termination conditions), and it would be difficult to fit them in to one of the existing frameworks for temporal abstraction. The PoE "basis" actions should be thought of as finding a smaller subset of "good" actions within a large space of possible actions.

5.2.2 Multiagent Language Learning

One interesting area of cooperative multiagent research is the idea of allowing individual agents to supplement their state information by communicating with one another. The communication channel between agents may be of limited bandwidth and noisy. The challenge is to learn a common "language". That is, the additional information that is presented by one agent must be interpreted by the others in a useful way. With enough bandwidth each agent could just present its state information to all others, turning the problem into a single large MDP (assuming that the agents collectively observe all state information). Of course this larger state space would slow learning. There is a trade-off between supplying additional information to improve the solution, and restricting this

information to speed learning. Language learning in co-operative multiagent systems has been explored in [Yanco and Stein 1993; Peshkin, Kim, Meuleau, and Kaelbling 2000; Jim and Giles 2001].

The latent variables in a product model can be thought of as just this kind of additional information shared between agents. However, in this case the information does not supplement the state information, but rather it replaces it. We could make the connection more explicit in the following way: First, add lateral connections between the state and action variables for each agent. Second, for each hidden unit, eliminate state-to-hidden connections from all but one of the agents' state variables. That is, only the state of one agent influences the latent variable for an expert. Now the hidden activities will consist of messages coming from individual agents. An agent will decide on an action based on its own state, and all hidden unit activities (See figure 5.3). Note that the direct state-to-action connections mean that the Boltzmann machine is no longer bipartite. However, if we always condition on a known state, then it is "conditionally bipartite". In this case exact inference is still tractable.

5.2.3 Conditional Completion

If the action allowed at a particular time step is constrained, it is natural to want to know what is the best action consistent with the constraints. For example, if one of the agents in the blocker task becomes unable to move in directions other than up, we would like to ask for actions for the other agents that are consistent with this restriction. Gibbs sampling allows us to do this easily by fixing a subset of action variables to their required values, and sampling the rest. The result is a set of good values for some action variables conditioned on the fixed values of the others (see figure 5.4).

Similarly, we can fix only some of the state variables, and sample others. No doubt this would be most useful for data completion: If a state variable is missing, it would be nice to fill it in with its most probable value, conditioned on the others. Unfortunately

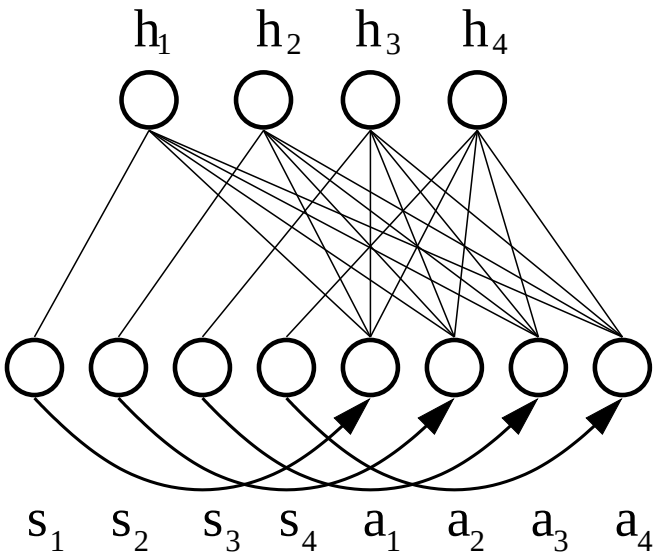

Figure 5.3: Connections for a language-interpretation of the hidden variables in a "conditionally bipartite" Boltzmann machine. This example shows the connections for a 4-agent system. The state variable for a particular agent connect directly to the corresponding action variable, and to the hidden "word" variable. All word variables are also connected to all action variables. Given the current state, inference in the Boltzmann machine is still tractable.

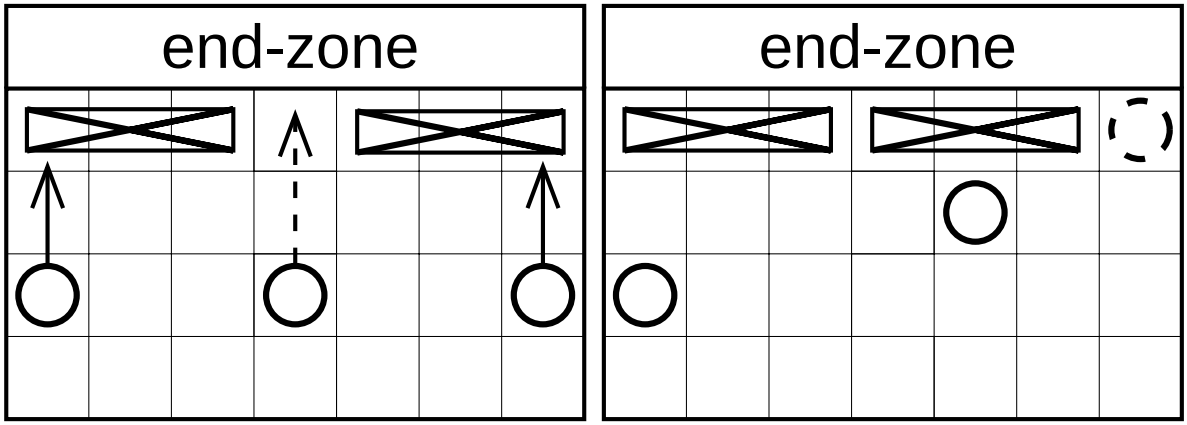

Figure 5.4: Action and state conditional completion. Good actions and optimistic or pessimistic states can be sampled conditioned on a subset of other states and actions.

we can not do this with a single value function network. Instead of filling in values according to how probable they are under the dynamics of the world, it will fill in values that yield high expected returns. In other words, the values that will be filled in for state variables will be those that are most desirable, not most probable. This could be used

for an optimistic form of state completion, giving an upper bound on what reward we expect to see given that we do not really know what values those state variables take on. Interestingly, we can also do pessimistic state completion: By reversing the sign of the free energy when sampling values for state variables, we can select values that will give us lower expected returns. In this way we get an estimate of how good or bad the current situation might be, and decide to act based on optimistic or pessimistic state completion. Alternatively we could learn a separate density model over states, and use it to do state completion according to the (learned) conditional distribution of state variables given other state variables.

5.3 Factored States and Actions

In the previous chapters I have presented methods for reinforcement learning in MDPs and POMDPs with factored state and action spaces. The methods can be combined in a number of ways.

We could use the POMDP method of chapter 3 to learn a DBN model of a POMDP, and use a real-valued PoE to model the value function in lieu of one of the other value function approximators. This would be useful if the action space was factored or real-valued.

An alternative would be to learn a dynamical model relating latent variables in the PoE model at different time steps (see figure 5.5). We could view this as learning an extended state that consists of the current observation and the latent variables at the previous time step. We would update the parameters of the PoE and the transition parameters so as to minimize the TD error across time steps. This is similar to the learning of additional memory bits in [Peshkin, Meuleau, and Kaelbling 1999], but in the context of value functions. The latent variables are treated as actions in the current time step, and additional state variables in the next step. Again, the hidden and action

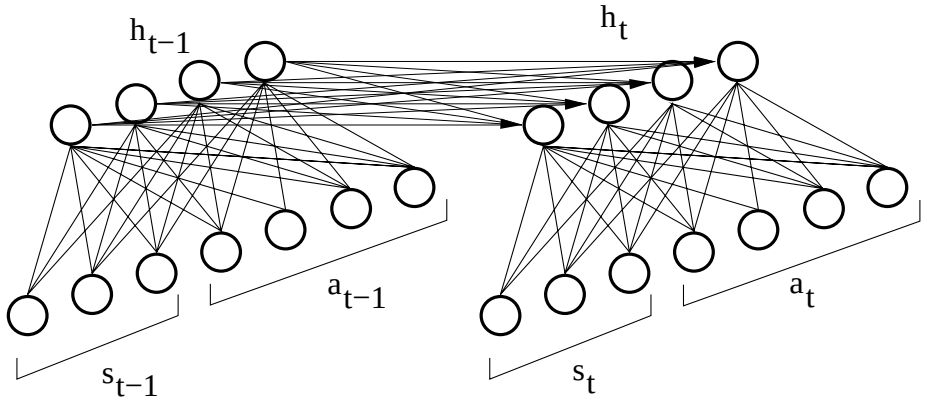

Figure 5.5: A dynamic PoE. The energy over hidden states at time t depend on the hidden states at time $t-1$ and the observation and action at time t.

variables could be drawn from a complex non-parameterized distribution dictated by the PoE model. Interestingly, this latter method can be related to learning task-specific DBN models (see section 5.4.3 below).

5.4 Future Work

There are a number of directions for future work. I outline some of the possibilities below.

5.4.1 Direct Policy Methods

Throughout this thesis I have concentrated on value-function-based solution methods. Another technique that has become popular is the "direct policy" approach [Williams 1992; Hansen 1998; Sutton, McAllester, Singh, and Mansour 2000; Ng and Jordan 2000]. The idea is that instead of first learning a value function and then deriving a policy from it, we should learn the parameters of a policy directly. Direct policy methods are interesting because in some cases the optimal value function can be quite hard to represent while the optimal policy is not.

The function approximators used in direct policy methods have typically been com-

pact mappings from states to distributions over actions. The PoE function approximation technique could be used in place of these explicit policy parameterizations. Instead of adjusting the parameters of the PoE so that the negative free energy approximates the value, the parameters would be adjusted so as to directly maximize the value of the induced policy. The policy would still be only implicitly parameterized, and extracted from the model with Gibbs sampling. This could be particularly advantageous in the cases where the state and action spaces are augmented with additional memory bits. The additional memory bits can be used to learn better policies for POMDPs [Peshkin, Meuleau, and Kaelbling 1999] or to pass information between co-operating agents [Peshkin, Kim, Meuleau, and Kaelbling 2000].

One way to encode correlations between such external memory bits in a direct policy method is to allow sequential setting of the bits. Bit one is set based on the current visible state, bit two is set based on the current visible state and bit one, and so on. This requires backward propagation of reward information through potentially many time steps. Another way is to pre-specify in the policy what correlations will be allowed. The idea is to make the correlations simple enough that the policy can be explicitly normalized. For example, we can specify fully-factored policies, or policies that allow correlations among small sets of action variables. Instead we can use a PoE network to implicitly parameterize the policy. This way we do not need to pre-specify which action variables can interact, or pass reward information backward through many steps of action-variable selection.

As an example, I trained a PoE network on a four-bit version of the stochastic policy task. Remember, for this task the best stochastic policy involves selecting from the two best actions with equal probability. The resultant performance was comparable to the value-function based method (see figure 5.6).

I used a very simple version of a direct policy method. The algorithm maintains a

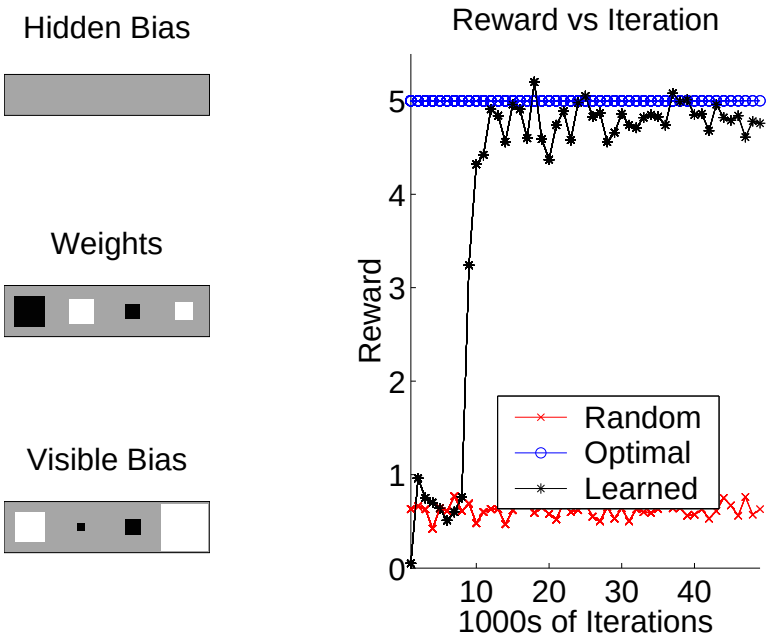

Figure 5.6: An RBM PoE network trained on a four-bit stochastic policy task. The network was trained using a simple direct-policy method that made "above average" actions more probable, and penalized "below average" actions.

decaying average of reward:

$$\overline{r}^{t+1} = 0.95\overline{r}^{t} + 0.05r^{t} \tag{5.2}$$

where $\overline{r}$ is the running average reward, and r is the instantaneous reward. The update rule decreased the free energy of an action if it was "above average" and increased it if it was "below average":

$$\Delta\theta \propto (\overline{r} - r)\,\frac{\partial F(\mathbf{a};\theta)}{\partial\theta} \tag{5.3}$$

So if an action was "above average", it was made more probable. If the action was "below average" it was made less probable. In this case the free energy does not correspond to value, but just captures something about relative goodness of actions.

This is a very simple "direct policy" method. Other methods have been developed that are theoretically sound and have proofs of convergence [Williams 1992; Hansen 1998; Sutton, McAllester, Singh, and Mansour 2000; Ng and Jordan 2000]. A PoE could be used with any of these methods, subject to one restriction: The method must require

only samples from the policy, not the policy distribution itself.

5.4.2 Hierarchical Learning

As we have seen in section 4.8.2 the hidden units can be interpreted as spatio-temporal macro actions. Interestingly, the hidden units do not tend to become active singly. Instead hidden units have a sparse but distributed activity pattern. It would be interesting to build a hierarchical model where the second layer could see only the visible states and the hidden units of the previous layer. The hidden units would then be interpreted as action variables by the higher-level model. Also, a POMDP model that treated the latent variables as extra state information (mentioned in section 5.3) could be organized hierarchically.

5.4.3 Task-Specific Distinctions

Currently the DBN model is learned by trying to maximize the likelihood of observations. While this results in the most generally applicable model, it is not necessarily the best model for solving a particular task. Given that the maximum size of the model is fixed in advance, it is possible that some statistical features of the data will be captured that are not relevant to the task. For example, there is a correlation between colour and object type in the driving task. This might be represented by the model even though colour is not really relevant to solving the driving task (at least, not when the "object type" attribute is available). It would be interesting to tie parameter learning more directly to learning the task. A simple first step in this direction would be to learn a model of immediate reward (see section 5.1). This would encourage the model to learn a representation where the immediate rewards were Markov in the current (core) state.

Alternatively we could use split/merge criteria that encourage the rewards to be Markov in the core states. This is the approach used by McCallum [1995]. On a similar vein, we could use ARD in the value function approximator to tell us which of the core

state variables are relevant to predicting future reward. If a particular variable is not informative about reward, the parameters linking it to the value function would fall to zero, and it could be pruned, irrespective of its role in the transition/emission model.

We could alter the learning rule for the model by penalizing it based on the size of the TD error. The TD error is a function of the current and future belief state. The future belief state is a deterministic function of the current belief state, action, and the next observation:

$$E_B^Q = r^t + \max_a Q(f(\phi^t, \mathbf{o}^t, a^t; \theta), a) - Q(\phi^t, a^t) \tag{5.4}$$

where f is the deterministic belief update function parameterized by θ. Given a fixed value function, we could penalize the update of a parameter of the transition or emission model by a factor proportional to the derivative of the TD error with respect to the parameter:

$$\Delta\theta \propto \frac{\partial}{\partial\theta} \left\langle \log P(\mathbf{s}^t, \mathbf{s}^{t+1}, \mathbf{o}^{t+1} | a^t) \right\rangle_{\hat{\phi}} - \frac{\partial}{\partial\theta} E_B^Q \tag{5.5}$$

This would encourage the model to learn a representation for which the value function approximation was a good approximator of the true value function.

Interestingly, we can draw a link between this variation of the DBN model and the "dynamic" version of the PoE method mentioned in section 5.3. Consider adding a parameter $\kappa \in [0, 1]$:

$$\Delta\theta \propto \kappa \frac{\partial}{\partial\theta} \left\langle \log P(\mathbf{s}^t, \mathbf{s}^{t+1}, \mathbf{o}^{t+1} | a^t) \right\rangle_{\hat{\phi}} - (1 - \kappa) \frac{\partial}{\partial\theta} E_B^Q \tag{5.6}$$

When $\kappa = 1$, we have the task-independent DBN model. When $\kappa = 0$ we have the task-specific dynamic PoE. We could try to learn the most general model that still meets performance requirements by slowly reducing κ over the course of learning while monitoring performance.

5.5 Summary

This thesis has shown that approximate inference, model learning and value function approximation allow for the solution of large factored Markov decision processes. The new techniques allow compact models and stochastic policies with unanticipated correlations to be learned. I have shown that the new algorithms can solve problems with extremely large state or action spaces (2^{180} bit states or 2^{40} bit actions). This thesis fills a gap at the intersection of model learning, approximate inference and reinforcement learning. It draws links between approximate inference, state representation and action selection. I have demonstrated new connections between the areas of graphical models and reinforcement learning that will hopefully stimulate further research. I have also made available a new set of test problems (available as Matlab functions) which can be used as benchmarks for future research.

The specific contributions of this thesis are:

1. A new method for modeling factored POMDPs. The DBN method combines approximate inference, model estimation and value-function approximation. I have demonstrated that the DBN algorithm can be used to solve POMDPs with very large state and observation spaces.

2. A new technique for approximating value functions for large factored MDPs and POMDPs. I have shown that the PoE algorithm can solve MDPs with very large action spaces, and can be used with both discrete and continuous MDPs.

3. A novel method for selecting actions based on the PoE function approximator. I have demonstrated that the brief sampling method works even for very large action spaces.

4. A new representation for stochastic policies on large action spaces. Because the policy is not explicitly represented, it is not necessary to know a priori exactly what

correlations should be included. As far as I am aware, this is the first method that is able to represent complex stochastic policies on large action spaces, when the policy may not have a compact functional representation.

5. Matlab implementations of the decision problems studied in the thesis. Hopefully these problems, particularly the Visual NY driving task, will encourage the development of other algorithms that can solve these new classes of large, interesting (PO)MDPs.

Along with these specific contributions, I have outlined areas of future research including multi-agent language learning, hierarchical action representation and task-specific model learning. This is certainly not the end of the story in the solution of large factored (PO)MDPs. In particular, although the methods here seem to work well in practice, there are no known convergence results or error bounds for them. I think that future research on learning task-specific models and directly learning stochastic policies might be particularly fruitful. Hopefully this thesis will encourage more cross-pollination between the areas of Bayesian inference, machine learning and reinforcement learning.

Appendix A

Task Functions

Where the variable type is not obvious, variables are generally described as either bit vectors, multinomial vectors, reals or structs (with a few exceptions). Bit vectors are Matlab vectors where each element is an integer in $\{0, 1\}$. Multinomial vectors are Matlab vectors where each element is an integer from an attribute-specific set. Reals are Matlab floating point numbers. Structs are task-specific Matlab structures. You generally do not need to know what is inside a given struct, since they can be initialized and manipulated through function calls.

Matlab code for all of the tasks below is available at
`http://www.gatsby.ucl.ac.uk/~sallans/tasks.html`.

A.1 New York Driving

The New York Driving task involves driving on a four-lane single-direction highway. The driver must pass slower cars and avoid getting in the way of faster cars.

The first function initializes the data structure that maintains the state of the simulator and the vector of observables.

```
% Initialize core (hidden) state and observation
[Sn,o]=NYinit()
Input:
  None

Output:
  Sn -- Initial core state (struct)
  o  -- Initial observation (multinomial vector)
```

The next function takes the current state of the simulator and an action and returns the next state of the simulator, a vector of observations and a reward.

```
% Step driving dynamics
[Sn,o,r]=NYstep(S,A)
Input:
  S -- Current core state (struct)
  A -- Current action

Output:
  Sn -- new core state (struct)
  o  -- emitted observation (multinomial vector)
  r  -- immediate reward (real)
```

The final functions display the current state and road in human-readable format.

```
% display state information
[]=displayNYState(o)
Input:
  o -- Current observation (multinomial vector)

Output:
  None

% display road state
[]=displayNYRoad(S)

Input:
  S -- Current core state (struct)

Output:
  None
```

A.1.1 Restricted NY Driving

This task is identical to the New York Driving task, except that the "gaze side" feature has been removed. Testing indicated that a memoryless policy performed very well on the original task. By removing the "gaze side" information, the task gained more hidden

state. The belief-state based policies continued to do well while the performance of the memoryless policy fell. The "masking" was done by always setting the "gaze side" attribute to "center" irrespective of the true value.

```
% Mask out gaze side
[oOut]=NYmask(oIn)
Input:
  oIn -- Current observation (multinomial vector)

Output:
  oOut -- new "masked" observation (multinomial vector)
```

A.1.2 Visual NY Driving

This task has the same state and reward structure as the NY driving task. However, in order to make the task more difficult, the sensory input was replaced with an 8×8 seven colour image and a single "horn" bit to indicate whether a faster car is stuck behind the driver.

In addition to the functions above there is a function to convert the current observable state to an image. This image, used in conjunction with the "hear horn" bit of the observation vector, was used for the Visual New York driving.

```
% generate image from observation
[I]=genNYPic(o)
Input:
  o -- current observation (multinomial vector)

Output:
  I -- 8x8 image (8x8 matrix with elements in [0,6])
```

A.2 Stochastic Policy

This task demonstrates that the free energy approach can learn and access stochastic policies where the correlations are not know a priori.

```
% Initialize core (hidden) state
[Sn]=stochasticInit(sa)
Input:
```

```
  sa -- size of action (number of bits)

Output:
  Sn -- Core state

% Step dynamics
[Sn,r]=stochasticStep(S,A)
Input:
  S -- Current core state (struct)
  A -- Current action (bit vector)

Output:
  Sn -- new core state (struct)
  r  -- immediate reward (real)
```

A.3 Blockers

This task shows that the free energy approach can be used to learn value functions for tasks with temporally delayed reward.

The first function initializes the states and actions of the simulator. The blockers are randomly placed along the top row and the agents are randomly placed along the bottom row.

```
% Initialize state
[Sn]=blockerInit(nA,nB,r,c)
Input:
  nA -- number of agents
  nB -- number of blockers
  r  -- number of rows in field
  c  -- number of columns in field

Output:
  Sn -- initialized state (bit vector)
```

The next function was used for the 5×4, two agents and one blocker task.

```
% Step blocker dynamics
[Sn,r]=blockerStep2(S,A)
```

```
Input:
  S -- Current state (bit vector)
  A -- Current action (bit vector)

Output:
  Sn -- new state (bit vector)
  r  -- immediate reward (real)
```

The final function was used for the 4×7, three agents and two blockers task. It can also be used for other numbers of agents, blockers and different field dimensions. The number of agents and blockers is computed by the function based on the state and action size, and the field dimensions.

```
% Step blocker dynamics
[Sn,r]=blockerStep3(S,A,r,c)
Input:
  S   -- Current state (bit vector)
  A   -- Current action (bit vector)
  r   -- number of rows in field
  c   -- number of columns in field

Output:
  Sn  -- new state (bit vector)
  rew -- immediate reward (real)
```

A.4 Large Action

The large action task shows that the free energy method can be used to represent value functions and search efficiently in large discrete state spaces.

```
% Initialize "key" states and actions
[kS]=largeActionInit(ss,as)
Input:
  ss -- length of state vector
  as -- length of action vector

Output:
  kS -- "key" state/action structure (struct)
```

```
% compute reward for a given state/action pair
[r]=largeActionStep(S,A,kS)
Input:
  S  -- current state (bit vector)
  A  -- Current action (bit vector)
  kS -- "key" state/action pairs (struct)

Output:
  r  -- immediate reward (real)
```

A.5 Rocket

The rocket task is used to show that the free energy method can be used to learn value functions over real-valued state and action spaces.

```
% Initialize rocket state
[Sn]=rocketInit();
Input:
  None

Output:
  Sn -- Rocket/target state (struct)

% Step rocket dynamics
[Sn,r]=rocketStep(S,A)
Input:
  S -- Current state (struct)
  A -- Current action (2x1 real vector)

Output:
  Sn -- new state (struct)
  r  -- immediate reward (real)
```

Bibliography

Ackley, D. H., G. E. Hinton, and T. J. Sejnowski (1985). A learning algorithm for Boltzmann machines. *Cognitive Science 9*, 147–169.

Baird, L. and H. Klopf (1993). Reinforcement learning with high-dimensional continuous actions. Technical Report WL-TR-93-1147, Wright Laboratory, Wright-Patterson Air Force Base.

Baird, L. and A. Moore (1999). Gradient descent for general reinforcement learning. See Kearns, Solla, and Cohn [1999].

Barto, A., S. Bradtke, and S. Singh (1995). Learning to act using real-time dynamic programming. *Artificial Intelligence 72*, 81–138.

Barto, A. G., R. S. Sutton, and C. W. Anderson (1983). Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man and Cybernetics 13*, 835–846.

Baum, L. E. and T. Petrie (1966). Statistical inference for probabilistic functions of finite state Markov chains. *Annals of Mathematical Statistics 41*.

Baum, L. E., T. Petrie, G. Soules, and N. Weiss (1970). A maximization technique occurring in the statistical analysis of probabilistic functions of Markov chains. *Annals of Mathematical Statistics 41*, 164–171.

Bellman, R. E. (1957a). *Dynamic Programming*. Princeton, NJ: Princeton University Press.

Bellman, R. E. (1957b). A Markov decision process. *Journal of Mathematical Mechanics 6*, 679–684.

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

Boutilier, C. and R. Dearden (1996). Approximating value trees in structured dynamic programming. In *ICML-96*.

Boutilier, C., R. Dearden, and M. Goldszmidt (1999). Stochastic dynamic programming with factored representations. Unpublished manuscript.

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

Boutilier, C. and D. Poole (1996). Computing optimal policies for partially observable decision processes using compact representations. In *Proc. AAAI-96*.

Boyen, X. and D. Koller (1998). Tractable inference for complex stochastic processes. In *Proc. UAI'98*.

Brown, A. D. and G. E. Hinton (2001). Products of hidden Markov models. In T. S. Jaakkola and T. Richardson (Eds.), *Proceedings of Artificial Intelligence and Statistics 2001*, pp. 3–11.

Cassandra, A. R. (1998). *Exact and Approximate Algorithms for Partially Observable Markov Decision Processes*. Providence, RI: Department of Computer Science, Brown University. Ph.D. thesis.

Cassandra, A. R., M. L. Littman, and N. L. Zhang (1997). Incremental pruning: A simple, fast, exact method for partially observable Markov decision processes. In *Uncertainty in Artificial Intelligence*.

Charnes, J. M. and P. P. Shenoy (1999). A forward Monte Carlo method for solving influence diagrams using local computation. School of Business Working Paper No. 273, School of Business, University of Kansas.

Chrisman, L. (1992). Reinforcement learning with perceptual aliasing: The perceptual distinctions approach. In *Tenth National Conference on AI*.

Cooper, G. F. (1988). A method for using belief networks as influence diagrams. In *Proc. of the Fourth Conference on Uncertainty in Artificial Intelligence*, pp. 55–63.

Cooper, G. F. (1990). The computational complexity of probabilistic inference using Bayesian belief networks. *Artificial Intelligence 42*, 393–405.

Dagum, P. and M. Luby (1993). Approximating probabilistic inference in Bayesian belief networks is NP-hard. *Artificial Intelligence 60*, 141–153.

Dayan, P. (1992). The convergence of TD(λ) for general λ. *Machine Learning 8*, 341–362.

Dean, T., R. Givan, and K. Kim (1998). Solving stochastic planning problems with large state and action spaces. In *Proc. Fourth International Conference on Artificial Intelligence Planning Systems*.

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

Dearden, R. (2001). Structured prioritized sweeping. In *ICML-2001*.

Dearden, R. and C. Boutilier (1997). Abstraction and approximate decision theoretic planning. *Artificial Intelligence 89*(1), 219–283.

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

Dietterich, T. (2000). Hierarchical reinforcement learning with the MAXQ value function decomposition. *Journal of Artificial Intelligence Research 13*, 227–303.

Doucet, A., S. Godsill, and C. Robert (2000). Marginal maximum a posteriori estimation using Markov chain Monte Carlo. Technical report, Cambridge University Department of Engineering.

Doya, K. (1996). Temporal difference learning in continuous time and space. See Touretzky, Mozer, and Hasselmo [1996], pp. 1073–1079.

Freund, Y. and D. Haussler (1992). Unsupervised learning of distributions on binary vectors using two layer networks. In J. E. Moody, S. J. Hanson, and R. P. Lippmann (Eds.), *Advances in Neural Information Processing Systems*, Volume 4. Morgan Kaufmann , San Mateo.

Geman, S. and D. Geman (1984). Stochastic relaxation, Gibbs distributions, and the Bayesian restoration of images. *IEEE Transactions on Pattern Analysis and Machine Intelligence 6*, 721–741.

Ghahramani, Z. and G. E. Hinton (1998). Variational learning for switching state-space models. *Neural Computation 4*(12), 963–996.

Ghahramani, Z. and M. I. Jordan (1997). Factorial hidden Markov models. *Machine Learning 29*, 245–273.

Glickman, M. and K. Sycara (2001). Evolutionary search, stochastic policies with memory, and reinforcement learning with hidden state. In *Proceedings of the Eighteenth International Conference on Machine Learning*.

Gordon, G. J. (2001). Reinforcement learning with function approximation converges to a region. See Leen, Dietterich, and Tresp [2001].

Guestrin, C., D. Koller, and R. Parr (2002). Multiagent planning with factored MDPs. In *Advances in Neural Information Processing Systems*, Volume 14. The MIT Press, Cambridge.

Hansen, E. A. (1998). Solving POMDPs by searching in policy space. In *Proc UAI'98*, pp. 211–219.

He, Q. and M. A. Shayman (1999). Solving POMDP by on-policy linear approximate learning algorithm. Technical report.

Hinton, G. E. (1999). Products of experts. In *ICANN-99*, Volume 1, pp. 1–6.

Hinton, G. E. (2000). Training products of experts by minimizing contrastive divergence. Technical Report GCNU TR 2000-004, Gatsby Computational Neuroscience Unit, UCL.

Hinton, G. E. and Z. Ghahramani (1997). Generative models for discovering sparse distributed representations. *Phil. Trans. Roy. Soc. London B 352*, 1177–1190.

Hinton, G. E. and T. J. Sejnowski (1986). *Parallel Distributed Processing*, Volume 1, Chapter Learning and Relearning in Boltzman Machines, pp. 282–317. The MIT Press, Cambridge.

Hinton, G. E. and Y.-W. Teh (2001). Discovering multiple constraints that are frequently approximately satisfied. In *Proc UAI 2001.*

Howard, R. A. (1960). *Dynamic Programming and Markov Processes*. Cambridge, MA: The MIT Press.

Howard, R. A. and J. E. Matheson (1984). *The Principles and Applications of Decision Analysis*, Volume II, Chapter Influence Diagrams, pp. 690–718. Strategic Decision Group, Mento Park, CA.

Jaakkola, T. S. (1997). *Variational Methods for Inference and Estimation in Graphical Models*. Cambridge, MA: Department of Brain and Cognitive Sciences, MIT. Ph.D. thesis.

Jaakkola, T. S., S. P. Singh, and M. I. Jordan (1995). Reinforcement learning algorithm for partially observable Markov decision problems. In G. Tesauro, D. S. Touretzky, and T. K. Leen (Eds.), *Advances in Neural Information Processing Systems*, Volume 7, pp. 345–352. The MIT Press, Cambridge.

Jim, K. and C. L. Giles (2001). How communication can improve the performance of multi-agent systems. In *Proc. Agents 2001.*

Jordan, M. I., Z. Ghahramani, T. S. Jaakkola, and L. K. Saul (1999). An introduction to variational methods for graphical models. *Machine Learning 37*, 183:233.

Kalman, R. E. (1960, March). A new approach to linear filtering and prediction problems. *Trans. ASME, Series D, Journal of Basis Engineering 82*, 35–45.

Kearns, M. S., S. A. Solla, and D. A. Cohn (Eds.) (1999). *Advances in Neural Information Processing Systems*, Volume 11. The MIT Press, Cambridge.

Koller, D. and R. Parr (1999). Computing factored value functions for policies in structured MDPs. In *Proc. IJCAI'99*.

Koller, D. and R. Parr (2000). Policy iteration for factored MDPs. In *Proc. of Uncertainty in AI (UAI-00), 2000.*

Konda, V. and J. Tsitsiklis (2000). Actor-critic algorithms. submitted to the SIAM Journal on Control and Optimization, February 2001.

Leen, T. K., T. Dietterich, and V. Tresp (Eds.) (2001). *Advances in Neural Information Processing Systems*, Volume 13. The MIT Press, Cambridge.

Littman, M. L., A. R. Cassandra, and L. P. Kaelbling (1995). Learning policies for partially observable environments: Scaling up. In *Proc. International Conference on Machine Learning*.

Littman, M. L., A. R. Cassandra, and L. P. Kaelbling (1996). Efficient dynamic-programming updates in partially observable Markov decision processes. Technical report CS-95-19, Department of Computer Science, Brown University.

Lovejoy, W. S. (1991). A survey of algorithmic methods for partially observable Markov decision processes. *Annals of Operations Research 28*, 47–66.

MacKay, D. J. C. (1994). Bayesian non-linear modeling for the energy prediction competition. *ASHRAE Transactions 100*(2), 1053–1062.

McAllester, D. A. and S. P. Singh (1999). Approximate planning for factored POMDPs using belief state simplification. In *Proc. UAI'99*.

McCallum, A. K. (1995). *Reinforcement learning with selective perception and hidden state*. Rochester NY: Dept. of Computer Science, Universiy of Rochester. Ph.D. thesis.

McCallum, R. (1997). Efficient exploration in reinforcement learning with hidden state. In *AAAI Fall Symposium on Model-directed Autonomous Systems*.

McGovern, A. (1998). acQuire-macros: An algorithm for automatically learning macro-actions. In *NIPS98 Workshop on Abstraction and Hierarchy in Reinforcement Learning*.

Meuleau, N., M. Hauskrecht, K.-E. Kim, L. Peshkin, L. P. Kaelbling, T. Dean, and C. Boutilier (1998). Solving very large weakly coupled Markov decision processes. In *Proc. AAAI-98*.

Monahan, G. E. (1982). A survey of partially observable Markov decision processes: Theory, models, and algorithms. *Management Science 28*, 1–16.

Neal, R. M. (1992). Connectionist learning of belief networks. *Artificial Intelligence 56*, 71–113.

Neal, R. M. (1993). Probabilistic inference using Markov chain Monte Carlo methods. Technical Report CRG-TR-93-1, Department of Computer Science, University of Toronto.

Neal, R. M. (1996). *Bayesian Learning for Neural Networks*. New York, NY: Springer Verlag.

Neal, R. M. (1998). Assessing relevance determination methods using DELVE. In C. M. Bishop (Ed.), *Neural Networks and Machine Learning*. New York, NY: Springer-Verlag.

Neal, R. M. (1999). Circularly-coupled Markov chain sampling. Technical Report No. 9910 Dept. of Statistics, University of Toronto.

Ng, A. Y. and M. I. Jordan (2000). PEGASUS: A policy search method for large MDPs and POMDPs. In *Proc UAI 2000*.

Ng, A. Y., R. Parr, and D. Koller (2000). Policy search via density estimation. See Solla, Leen, and Müller [2000], pp. 1022–1028.

Parr, R. and S. Russell (1995). Approximating optimal policies for partially observable stochastic domains. In *Proc. IJCAI-95*.

Parr, R. and S. Russell (1998). Reinforcement learning with hierarchies of machines. In M. I. Jordan, M. J. Kearns, and S. A. Solla (Eds.), *Advances in Neural Information Processing Systems*, Volume 10. The MIT Press, Cambridge.

Pearl, J. (1988). *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. San Mateo, CA: Morgan Kaufmann.

Peng, J. and R. L. Williams (1994). Incremental multi-step Q-learning. In *Proc.* 11^{th} *International Conference on Machine Learning*, pp. 226–232.

Peshkin, L., K.-E. Kim, N. Meuleau, and L. P. Kaelbling (2000). Learning to cooperate via policy search. In *Proc. UAI 2000*.

Peshkin, L., N. Meuleau, and L. P. Kaelbling (1999). Learning policies with external memory. In *Proc. 16th International Conference on Machine Learning*, pp. 307–314.

Poupart, P. and C. Boutilier (2000). Value-directed belief state approximation for POMDPs. In *Proc. UAI 2000*.

Poupart, P. and C. Boutilier (2001). Vector-space analysis of belief-state approximation for POMDPs. In *Proc UAI 2001.*

Poupart, P., L. E. Ortiz, and C. Boutilier (2001). Value-directed sampling methods for monitoring POMDPs. In *Proc UAI 2001.*

Precup, D., R. Sutton, and S. Singh (1998). Theoretical results on reinforcement learning with temporally abstract options. In *Proceedings of the Tenth European Conference on Machine Learning*.

Putterman, M. (1994). *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. New York: Wiley.

Rabiner, L. R. and B.-H. Juang (1986, January). An introduction to hidden Markov models. *IEEE ASSAP Magazine 3*, 4–16.

Rodriguez, A., R. Parr, and D. Koller (2000). Reinforcement learning using approximate belief states. See Solla, Leen, and Müller [2000].

Rumelhart, D. E., G. E. Hinton, and R. J. Williams (1986). Learning internal representations by back-propagating errors. *Nature 323*, 533–536.

Rummery, G. A. and M. Niranjan (1994). On-line Q-learning using connectionist systems. Technical Report CUED/F-INFENG/TR 166, Engineering Department, Cambridge University.

Russell, S. (1998). Learning agents for uncertain environments (extended abstract). In *Proc. COLT-98*. ACM Press.

Sabes, P. N. and M. I. Jordan (1996). Reinforcement learning by probability matching. See Touretzky, Mozer, and Hasselmo [1996], pp. 1080–1086.

Sallans, B. (1998). *A Hierarchical Community of Experts*. Toronto, Canada: University of Toronto. M.Sc. thesis.

Sallans, B. (2000). Learning factored representations for partially observable Markov decision processes. See Solla, Leen, and Müller [2000], pp. 1050–1056.

Sallans, B. and G. E. Hinton (2001). Using free energies to represent Q-values in a multiagent reinforcement learning task. See Leen, Dietterich, and Tresp [2001].

Santamaria, J., R. Sutton, and A. Ram (1998). Experiments with reinforcement learning in problems with continuous state and action spaces. *Adaptive Behavior 6*(2).

Sato, M. and S. Ishii (1999). Reinforcement learning based on on-line EM algorithm. See Kearns, Solla, and Cohn [1999].

Saul, L. K., T. S. Jaakkola, and M. I. Jordan (1996). Mean field theory for sigmoid belief networks. *Journal of Artificial Intelligence Research 4*, 61–76.

Schneider, J., W.-K. Wong, A. Moore, and M. Riedmiller (1999). Distributed value functions. In *Proc. 16th International Conf. on Machine Learning*, pp. 371–378. Morgan Kaufmann, San Francisco, CA.

Shachter, R. D. (1986). Evaluating influence diagrams. *Operations Research 34*, 871–882.

Shachter, R. D. and C. R. Kenley (1989). Gaussian influence diagrams. *Management Science 35*, 527–550.

Shachter, R. D. and M. A. Peot (1992). Decision making using probabilistic inference methods. In *Proc. of the Eighth Conference on Uncertainty in Artificial Intelligence*, pp. 276–283.

Shimony, S. E. (1994). Finding maps for belief networks is NP-hard. *Artificial Intelligence 68*, 399–410.

Shumway, R. H. and D. S. Stoffer (1982). An approach to time series smoothing and forecasting using the EM algorithm. *Journal of Time Series Analysis 3*(4), 253–264.

Singh, S. P. and P. Dayan (1998). Analytical mean squared error curves in temporal difference learning. *Machine Learning 32*, 5–40.

Singh, S. P. and R. S. Sutton (1996). Reinforcement learning with replacing eligibility traces. *Machine Learning 22*, 123.

Smallwood, R. D. and E. J. Sondik (1973). Optimal control of partially observable processes over the finite horizon. *Operations Research 21*, 1071–1088.

Smolensky, P. (1986). Information processing in dynamical systems: Foundations of harmony theory. In D. E. Rumelhart and J. L. McClelland (Eds.), *Parallel Distributed Processing: Explorations in the Microstructure of Cognition. Volume 1: Foundations*. Cambridge, MA: The MIT Press.

Solla, S. A., T. K. Leen, and K.-R. Müller (Eds.) (2000). *Advances in Neural Information Processing Systems*, Volume 12. The MIT Press, Cambridge.

Sondik, E. J. (1973). The optimal control of partially observable Markov processes over the infinite horizon: Discounted costs. *Operations Research 26*, 282–304.

Sondik, E. J. (1976). *The Optimal Control of Partially Observable Markov Processes*. Stanford, Calf.: Department of Engineering-Economic Systems, Stanford University. Ph.D. thesis.

St-Aubin, R., J. Hoey, and C. Boutilier (2000). Apricodd: Approximate policy construction using decision diagrams. See Solla, Leen, and Müller [2000].

Stolcke, A. and S. Omohundro (1993). Hidden Markov Model induction by Bayesian model merging. In S. J. Hanson, J. D. Cowan, and C. L. Giles (Eds.), *Advances in Neural Information Processing Systems*, Volume 5, pp. 11–18. Morgan Kaufmann, San Mateo.

Sutton, R. S. (1988). Learning to predict by the method of temporal differences. *Machine Learning 3*, 9–44.

Sutton, R. S. (1990). Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *Proc. International Conference on Machine Learning*.

Sutton, R. S. (1996). Generalization in reinforcement learning: Successful examples using sparse coarse coding. See Touretzky, Mozer, and Hasselmo [1996], pp. 1038–1044.

Sutton, R. S. and A. G. Barto (1998). *Reinforcement Learning: An Introduction*. Cambridge, MA: The MIT Press.

Sutton, R. S., D. McAllester, S. P. Singh, and Y. Mansour (2000). Policy gradient methods for reinforcement learning with function approximation. See Solla, Leen, and Müller [2000], pp. 1057–1063.

Tadepalli, P. and D. Ok (1996). Scaling up average reward reinforcement learning by approximating the domain models and the value function. In *Proc. 13th International Conference on Machine Learning*.

Teh, Y.-W. and G. E. Hinton (2001). Rate-coded restricted boltzmann machines for face recognition. See Leen, Dietterich, and Tresp [2001].

Thrun, S. (2000). Monte Carlo POMDPs. See Solla, Leen, and Müller [2000].

Touretzky, D. S., M. C. Mozer, and M. E. Hasselmo (Eds.) (1996). *Advances in Neural Information Processing Systems*, Volume 8. The MIT Press, Cambridge.

Watkins, C. J. C. H. (1989). *Learning from Delayed Rewards*. Cambridge, UK: Cambridge University. Ph.D. thesis.

Watkins, C. J. C. H. and P. Dayan (1992). Q-learning. *Machine Learning 8*, 279–292.

Williams, R. J. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning 8*, 229–256.

Yanco, H. and L. Stein (1993). An adaptive communication protocol for cooperating mobile robots. In J.-A. Meyer, H. Roitblat, and S. Wilson (Eds.), *From Animals to Animats: International Conference on Simulation of Adaptive Behavior*, pp. 478–485. The MIT Press.

Zhang, N. L. (1998). Probabilistic inference in influence diagrams. In *Proc. UAI'98*, pp. 514–522.

Zhang, N. L. and W. Liu (1996). Planning in stochastic domains: Problem characteristics and approximation. Technical report HKUST-CS96-31, Department of Computer Science, Hong Kong Univeristy of Science and Technology.