SCALING REINFORCEMENT LEARNING TO THE UNCONSTRAINED MULTI-AGENT DOMAIN

A Dissertation

by

VICTOR PALMER

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

August 2007

Major Subject: Computer Science

SCALING REINFORCEMENT LEARNING TO THE UNCONSTRAINED MULTI-AGENT DOMAIN

A Dissertation

by

VICTOR PALMER

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Approved by:

Chair of Committee,	Thomas Ioerger
Committee Members,	Brit Grosskopf
	John Keyser
	Vivek Sarin
Head of Department,	Valerie Taylor

August 2007

Major Subject: Computer Science

ABSTRACT

Scaling Reinforcement Learning to the Unconstrained Multi-Agent Domain.

(August 2007)

Victor Palmer, B.S., Lubbock Christian University

Chair of Advisory Committee: Dr. Thomas Ioerger

Reinforcement learning is a machine learning technique designed to mimic the way animals learn by receiving rewards and punishment. It is designed to train intelligent agents when very little is known about the agent's environment, and consequently the agent's designer is unable to hand-craft an appropriate policy. Using reinforcement learning, the agent's designer can merely give reward to the agent when it does something right, and the algorithm will craft an appropriate policy automatically. In many situations it is desirable to use this technique to train systems of agents (for example, to train robots to play RoboCup soccer in a coordinated fashion). Unfortunately, several significant computational issues occur when using this technique to train systems of agents. This dissertation introduces a suite of techniques that overcome many of these difficulties in various common situations.

First, we show how multi-agent reinforcement learning can be made more tractable by forming coalitions out of the agents, and training each coalition separately. Coalitions are formed by using information-theoretic techniques, and we find that by using a coalition-based approach, the computational complexity of reinforcement-learning can be made linear in the total system agent count. Next we look at ways to integrate domain knowledge into the reinforcement learning process, and how this can significantly improve the policy quality in multi-agent situations. Specifically, we find that integrating domain knowledge into a reinforcement learning process can overcome

training data deficiencies and allow the learner to converge to acceptable solutions when lack of training data would have prevented such convergence without domain knowledge. We then show how to train policies over continuous action spaces, which can reduce problem complexity for domains that require continuous action spaces (analog controllers) by eliminating the need to finely discretize the action space. Finally, we look at ways to perform reinforcement learning on modern GPUs and show how by doing this we can tackle significantly larger problems. We find that by offloading some of the RL computation to the GPU, we can achieve almost a 4.5 speedup factor in the total training process.

To Andy

ACKNOWLEDGMENTS

The Fannie and John Hertz Foundation has been extraordinarily generous with their financial support throughout my graduate career. Without the Foundation's support, my graduate experience would simply not have happened. I am grateful beyond words to the Foundation and its directors.

TABLE OF CONTENTS

CHAPTER — Page

I INTRODUCTION 1

A. Reinforcement Learning Background 2
1. History of Reinforcement Learning 3
2. Mathematical Notation 6
3. Review of Reinforcement Learning Techniques 7
a. Value Iteration 7
b. Policy Iteration 8
c. Model-Free Methods: Q-Learning 8
d. Least-Squares Policy Iteration 9
B. Reinforcement Learning in Multi-Agent Systems 12
C. Related Research 13
1. Naive Single-Agent Training 14
a. Relation to Game Theory 15
b. Relation to Gradient Ascent 15
2. Coordinated Reinforcement Learning 17
3. Modified Reward Functions 18
4. TPOT-RL 19
D. Extending Reinforcement Learning to Unconstrained Multi-Agent Domains 19

II REDUCING REINFORCEMENT-LEARNING COMPLEXITY THROUGH COALITION FORMATION 23

A. Introduction 23
B. Contribution 26
1. Classifier Training and Feature Selection 26
a. Wrappers 27
b. Filters 27
2. Approach 28
C. Preliminaries 29
1. Coalitions and Spaces 29
2. Reduced Training Corpi 30
3. Policies 31
4. Coalition Structure Valuation 31

CHAPTER Page

5. Choice of Reward Function 32
D. Information Theory 33
1. Introduction 33
2. Application to Reinforcement Learning 35
E. Why Maximize Mutual Information? 37
1. IBCF Algorithm 42
F. Analysis of Computational Requirements 42
1. Introduction 42
2. Analysis 43
3. Discussion 44
G. Errors and the Information Integral 45
H. Error and Policy Quality 49
1. Coalitions and the Markov Property 50
2. Deviations 50
3. A More Stringent Requirement 51
4. Discussion 56
I. Experiment 1: Multi-Agent Cart-Pole Balancing Problem . 57
J. Experiment 2: Multi-Agent Network Control Problem (SysAdmin) 61
1. Discussion 71
K. Experiment 3: Power Distribution 73
L. Conclusions 74

III INTEGRATING FUZZY KNOWLEDGE INTO REINFORCEMENT LEARNING 76

A. Sample Complexity and Reinforcement Learning 78
B. Domain Knowledge and Fuzzy Rules 79
C. Potential Negative Effects of Added Domain Knowledge . . 80
1. Derivation 80
2. Interpretation 84
D. Fuzzy Reward Shaping 86
E. Integrating Fuzzy Knowledge 87
F. Cart-Pole Balancing Problem Domain 88
G. Conclusions 90

IV LEAST-SQUARES POLICY ITERATION OVER CONTINUOUS ACTION SPACES 94

A. Introduction 94

CHAPTER Page

B. Related Work . . . 95
C. Continuous-Action Selection Failure on Narrow-Q Domains 96
1. Basis Functions . . . 96
2. Symmetry-Induced Problems with Action Selection . . 99
3. A Practical Example of an Extreme Narrow-$\hat{Q}$ Domain 99
4. Action Selection: The Problem . . . 100
5. Solutions . . . 103
6. Heuristic for Transforming Narrow-Q Domains 104
7. Reducing the Heuristic Error . . . 108
D. Experiment 1: Continuous Pole Balancing . . . 108
1. Results . . . 112
E. Stochastic Action Selection . . . 116
1. Simulated Annealing-Based Action Selection 118
2. Experiment 2: Simulated Annealing Action Selection . 118
3. Exploiting Temporal Coherence . . . 120
4. Experiment 3: Simulated Annealing Action Search with Temporal Coherence . . . 120
F. Conclusions . . . 121

V ACCELERATED APPROXIMATE REINFORCEMENT LEARNING ON THE GPU . . . 123

A. Previous Work / Background . . . 124
1. Parallel Computation in Artificial Intelligence 124
2. GPU Background . . . 125
B. Approach . . . 125
1. Factoring **A** . . . 127
C. Basis Functions in LSPI . . . 128
D. Brook Architecture . . . 131
1. Streams . . . 131
2. Kernels and Reductions . . . 132
E. Description of the Algorithm . . . 133
1. Step 1: Loading the Experience Tuples . . . 133
2. Step 2: Start Calculating the 'Post' Actions 137
3. Step 3: Action Determination . . . 140
4. Step 4: Computing Pre and Post Basis Functions . . . 142
5. Step 5: Computing **A** . . . 144
F. Experiment . . . 146
1. Results . . . 148

CHAPTER Page

2. Discussion 151
G. Conclusion 151

VI CONCLUSION 153

A. Final Thoughts 156

REFERENCES 158

APPENDIX A 171

VITA 192

LIST OF TABLES

TABLE Page

I Computational complexity of common RL algorithms 13

II Results of the cart-pole balancing experiment, $\pm$ one STD shown . . 60

III Algorithm performance on the network control problem as a function of connected nodes 67

IV Performance comparisons on the 12-node, 2 connection set 69

V Average training times (minutes) 72

VI Results of the power-grid experiment 73

VII Rules integrated into the inverted pendulum problem 89

VIII Number of training examples before the learner had a 90% success probability 92

IX Agent simulation settings 112

X Upright timesteps (capped at 1000) for the various simulation environments (one STD shown) 117

XI Search steps with and without the consideration of temporal coherence 121

LIST OF FIGURES

FIGURE Page

1 The cart-pole balancing problem . . . 57

2 Relative information example . . . 59

3 Algorithm comparison with fixed network connectivity . . . 60

4 Example network topologies . . . 62

5 Algorithm performance as a function of network connectivity (number of connections per node) . . . 66

6 Results of the SysAdmin simulation . . . 68

7 Average training times (in minutes) for the various algorithms as a function of network size. . . . 70

8 The cart-pole balancing problem revisited . . . 87

9 Fuzzy membership functions illustrated . . . 89

10 The results of the sample complexity experiment . . . 91

11 The inverted pendulum balancing problem . . . 97

12 Illustration of the basis functions used in the IP problem . . . 98

13 Illustration of the Narrow-Q problem . . . 102

14 Approximation error and various basis functions . . . 109

15 Results for the "Experiment 1" set of experiments . . . 113

16 Force response diagram for the 2-action strong agent . . . 115

17 Force response diagram for the continuous-action agent . . . 116

18 Results for the "Experiment 2" set of experiments . . . 119

FIGURE Page

19 Force diagrams for the cart-pole balancing problem 128

20 The cart-pole balancing problem revisited 129

21 Step 1 of the LSPI algorithm in Brook 134

22 Step 2 of the LSPI algorithm in Brook 138

23 Step 3 of the LSPI algorithm in Brook 141

24 Step 4 of the LSPI algorithm in Brook. 143

25 Step 5 of the LSPI algorithm in Brook 145

26 Benchmark of the Brook implementation of the LSPI algorithm . . . 149

27 Second benchmark of the Brook implementation of the LSPI algorithm 150

CHAPTER I

INTRODUCTION

Agents are extraordinarily powerful constructs. Once trained, they are capable of adaptive, autonomous action, and have proved themselves to be useful in a vast array of diverse environments. In fact, recent years have seen an explosion in the number of successful agent-based system that have been significantly deployed. Application domains have run the gamut of everything from air-traffic control [1], Internet news (Newt) and mail (Maxims) filtering [2], electronic commerce (Kabash) [3], business process managment (ADEPT) [4], medical patient monitoring (Guardian) [5] and managemenet [6], AI for video game opponents (reactive agents)[7], etc.

Of course, each of these applications represents a great deal of work by humans to craft agent behavior in such a way that the agent designer's goals are accomplished. To make this task a *little* easier, there are a variety of agent programming languages (historically everything from low-level languages such as Prolog or Lisp, to very high-level cognitive modeling tools such as ACT-R [8, 9] and SOAR have been used [10]) that can help streamline the process of crafting an agent's behavior (also called an agent's *policy*). Even so, it is not very difficult to imagine that in some cases, the complexity of crafting an agent's behavior can be quite extreme, especially when trying to design policies for entire systems of agents (for example, the coordination mechanisms for robotic soccer teams in RoboCup [11, 12]).

An alternative to programming agent behavior by hand is to use various forms of machine learning that allow agents to learn on their own how to act in different circumstances. In some cases this approach is absolutely required: sometimes, al-

---

The journal model is Artificial Intelligence.

though an agent's creator knows *what* he wants the agent to do, he has no idea *how* the agent should act to accomplish that goal).

More generally, when using these machine learning techniques, one specifies a *task* or *goal* that one wants an agent to achieve, and then lets the algorithm automatically sculpt an agent's policy to achieve the goal or task. Fortunately, today there are a variety of algorithms that are capable of this type of automatic policy generation, including genetic algorithms [13, 14, 15], neural approaches [16], and reinforcement learning [17, 18], among others. In this dissertation we will focus on the latter of these techniques. Specifically, we will concentrate on some of the issues encountered when trying to apply reinforcement learning to systems of multiple agents.

Practically there are several major computational issues that prevent reinforcement learning from being applied in this type of multi-agent environment. Primarily these issues are computational in nature. That is, most reinforcement learning algorithms have computational requirements which are simply untenable when applied to multi-agent domains. In the next sections we will introduce and formalize some of these problems, and then, in the rest of this dissertation, propose solutions to address some of these problems.

A. Reinforcement Learning Background

The term "reinforcement learning" (RL) [17, 18] is broadly used to describe the process of training an agent to choose its actions in order to maximize some measure of reward. This learning usually takes place in an environment with an *a priori* unknown structure such that the agent must learn how to act by repeated interactions with the environment.

RL is especially useful in domains where it is not clear exactly how an agent's creator would hand-code a good policy. Modern uses of reinforcement learning include some very interesting and *hard* problems such as developing effective treatment schedules for HIV patients [19], building natural-spoken-language human-computer interfaces [20], imagine recognition [21], balancing and bicycle riding [22, 23, 24], robot coordination [25], robot swimming [26], and simulating various neurobiological phenomena such as the auditory system [27] and temporal sequence learning [28].

1. History of Reinforcement Learning

Historically, reinforcement learning represents the merger of two related fields of study. The first field grew out of a study called "optimal control", a somewhat out-dated term (mainly used in the 1950s) to describe the problem of developing a controller to minimize some aspect of a dynamical system's operation (perhaps the amount of energy expended by a manufacturing plant controller, for example). This field actually has roots which reach as far back as Hamilton's physics-based study of dynamic systems, but in 1957 it was modernized by Richard Bellman [29] and recast into the field of study now called "dynamic programming". Bellman's contribution was at least two-fold: first, he recast the "optimal control" problem in discrete stochastic terms (in terms of a Markov Decision Process). Traditionally, the "optimal control" problem was phrased in continuous terms since it was generally applied to real, physical dynamic systems. Second, he introduced an algorithm called value iteration (which we will introduce in detail later) to solve the "optimal control problem" on discrete stochastic domains. Shortly thereafter, an almost identical algorithm was proposed by Howard [30], and modern times have seen an explosion in related algorithms: approximation methods (see [31] for a summary), asynchronous methods [32, 33],

situations where state information is limited (POMDPs, see [34] for a summary), etc.

The second field of study that eventually merged into reinforcement learning originated in the psychological community with the study of trial and error learning. There is a famous quote from Thordike (one of the pioneers of this field), which Thorndike called the "Law of Effect":

> Of several responses made to the same situation, those which are accompanied or closely followed by satisfaction to the animal will, other things being equal, be more firmly connected with the situation, so that, when it recurs, they will be more likely to recur; those which are accompanied or closely followed by discomfort to the animal will, other things being equal, have their connections with that situation weakened, so that, when it recurs, they will be less likely to occur. The greater the satisfaction or discomfort, the greater the strengthening or weakening of the bond. (Thorndike, 1911, p. 244)

The quotation rings almost eerily true when one thinks of how modern reinforcement algorithms operate, especially algorithms such as Watkins's Q-Learning [35, 36] (discussed in more detail shortly). More importantly Thorndike's Law of Effect described a learning organism that interacted with its environment and taught itself how to act to receive 'satisfaction'.

With the advent of research on artificial intelligence, AI researchers began to experiment with ways to implement this trial-and-error type learning algorithmically. Perhaps some of the earliest research into trial-and-error learning was done almost simultaneously by two separate groups in 1954. In that year Minsky [37] described constructing an analog (effectively neural-net-based) machine designed to learn by trial and error (essentially mimicking the neuro-biological Hebb's rule [38], which had

been introduced only a few years earlier), and Farley and Clark came up with an almost identical proposal that same year [39]. Minsky also made a huge contribution in 1961 when he (probably for the first time) formalized the credit-assignment problem: 'if one receives a reward *now*, how does one know which past actions helped to obtain that reward?'. In many ways, almost all reinforcement learning algorithms try to solve this exact problem [40].

Concurrently, some fundamental applications of reinforcement learning materialized. Perhaps the first was Michie's designing of a *colored bead-based* machine called MENACE (Matchbox Educable Noughts and Crosses Engine) that could learn how to play tic-tac-toe [41, 42], and another algorithm called BOXES which successfully tackled the pole-balancing problem [43]. The game of Blackjack was well-learned by reinforcement methods soon after [44]. Also worthy of mention here, though developed much earlier, is Samuel's famous checkers program [45], although by appearances Samuel worked very much independently of the rest of the reinforcement learning tract during this period.

Finally, the two threads of research, psychologically-based trial-and-error learning and the optimal-control problem merged with the introduction of Watkin's Q-learning algorithm in 1989 [36]. This algorithm was based on an agent exploring its environment and developing an appropriate policy that would generate high levels of reward, but also, contained rigorous proofs that the policies generated by this mechanism *also* solved the optimal control problem in many domains. The introduction of this algorithms was soon followed by Tesauro's development of a highly-successful backgammon program, TD-Gammon in 1992 [46]. More modern times have seen the introduction of *approximate reinforcement learning algorithms* [23], and the use of neural-nets to perform RL [25].

2. Mathematical Notation

Before we proceed with our introduction to reinforcement learning, it will be extremely helpful to nail down the mathematical terminology. Mathematically, in this dissertation we will work in the typical (multi-agent) reinforcement learning paradigm [17] in which a set of learning agents interacts with an environment modeled as a Markov decision process (MDP). Specifically, let there be a set of N agents $\{n_1, n_2, \ldots, n_N\}$ which interacts with an environment modelled by an MDP. At each time-step the environment is in some state $s_i \in \mathcal{S}$, and each agent can perform an action $a_i \in \mathcal{A}$. As such the environment state, joint agent actions and reward at each time-step $t \in \{0, 1, 2, \ldots\}$ are denoted $s_t \in \mathcal{S}$, $\mathbf{a_t} \in \mathcal{A}^N$, and $r_t \in \mathcal{R}$ respectively. Since the environment is modelled as an MDP, its dynamics are characterized by state transition probabilities, $\mathcal{P}(s, \mathbf{a}, s') = Pr\{s_{t+1} = s' | s_t = s, \mathbf{a_t} = \mathbf{a}\}$, and expected rewards $\mathcal{R}(s, \mathbf{a}) = E\{r_{t+1} | s_t = s, \mathbf{a_t} = \mathbf{a}\}$, which we assume to be finite.

System agents decide what actions to take at each time-step by following a policy $\pi(\mathbf{a_t}; s_t)$ such that the probability that the system agents will (collectively) perform action $\mathbf{a_t}$ when the environment is in state s_t is $\pi(\mathbf{a_t}; s_t)$. Under a given policy, one can define the expected discounted reward $Q^\pi(s, a)$ an agent would receive if it started in state s, performed action a, and then followed policy π:

$$Q^\pi(s, a) = E_{a_t \sim \pi; s_t \sim \mathcal{P}} \left[\sum_{t=0}^{\infty} \gamma^t r_t \mid s_0 = s, a_0 = a \right] \tag{1.1}$$

For any such system, there exists an optimal policy π^* such that:

$$\forall s, a \;\; Q^{\pi^*}(s, a) \geq \max_\pi Q^\pi(s, a) \tag{1.2}$$

Other measures of policy quality exist, such as the average policy reward $\eta(\pi)$:

$$\eta(\pi) = \sum_{s,a} \rho^{\pi}(s)\pi(a;s)R(s,a) \tag{1.3}$$

3. Review of Reinforcement Learning Techniques

Now that we have a mathematical foundation for talking about reinforcement learning, we can survey several contemporary RL algorithms in more detail:

a. Value Iteration

Value iteration [47, 29] is a straightforward method which attempts to directly learn the optimal discounted reward of a given state-action pair (or more simply, learn the $Q^{\pi^*}(s,a)$ function). The optimal policy can then be recovered simply as:

$$\pi^*(s) = \arg\max_a Q^{\pi^*}(s,a) \tag{1.4}$$

This algorithm proceeds by performing the following update infinitely often (in practice until the Q-values converge):

$$Q(s,a) \Leftarrow R(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s,a,s')Q(s',\pi(s')) \tag{1.5}$$

$$\pi(s) = \arg\max_a Q(s,a) \tag{1.6}$$

This update can be shown to converge such that $\pi \rightarrow \pi^*$ [29, 47]. The advantage of value iteration is that it is very flexible (the updates can be performed asynchronously and in parallel [48]), though it is relatively slow to converge, and in the worst case, the number of required iterations before the optimal policy is reached

grows polynomially in $1/(1-\gamma)$ such that convergence is extremely slow for systems with a discounting factor ≈ 1 [49]. As such, value iteration is rarely used on the sizeable problems of interest today.

b. Policy Iteration

Policy iteration [49] works by directly searching through policy space for the optimal policy π^*. The algorithm begins by choosing an arbitrary initial policy π and then performing the following steps:

Step 1: Solve the linear equations:

$$V_\pi(s) = R(s, \pi(s)) + \gamma \sum_{s' \in \mathcal{S}} P(s, \pi(s), s') V_\pi(s') \tag{1.7}$$

Step 2: Improve the policy:

$$\pi(s) \Leftarrow \arg\max_a \ (R(s, a) + \gamma \sum_{s' \in \mathcal{S}} P(s, a, s') V_\pi(s')) \tag{1.8}$$

Since there are at most $|\mathcal{A}|^{|\mathcal{S}|}$ unique policies, and the sequence of policies generated by policy iteration have increasingly higher valuations, the algorithm terminates with at the optimal policy in (at worst) an exponential number of iterations [50].

c. Model-Free Methods: Q-Learning

In domains where we do not have access to the transition function $P(s, a, s')$, we can still search for the optimal policy by using a technique called Q-learning [36, 35]. In

Q-learning, an agent explores the state/action space (either by random exploration or by some other heuristic) and repeatedly performs the update:

$$Q(s,a) \Lleftarrow Q(s,a) + \alpha(r + \gamma \max_{a'} Q(s',a') - Q(s,a)) \tag{1.9}$$

where $\langle s,a,r,s' \rangle$ is an experience tuple gathered from state space exploration, and α is a learning rate. It has been shown that under this update rule the $Q(s,a)$ will converge to $Q^*(s,a)$ (under the condition that the update is performed an infinite number of times) [51, 52].

d. Least-Squares Policy Iteration

State spaces in modern RL problems can be absolutely huge, and as such explicit representation of the Q-function is rarely an option. Fortunately, it is often times possible to approximate a Q-function as a linear superposition of basis functions ϕ_i such that (for some parameter vector $\omega \in \mathbb{R}^{\mathbf{k}}$):

$$Q(s,a) \approx \sum_{i=1}^{k} \phi_i(s,a)\omega_i = \hat{Q}(s,a;\omega) \tag{1.10}$$

For a given policy π, we can determine the ω such that:

$$\sum_{s,a} \left| Q^{\pi}(s,a) - \hat{Q}(s,a;\omega) \right| \tag{1.11}$$

is minimized by using Least-Squares Temporal Difference Q-Learning (LSTD-Q) [23]. We can rewrite the expression for $Q(s,a)$:

$$R(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s,a,s') \sum_{a' \in \mathcal{A}} \pi(a';s') Q^{\pi}(s',a') \quad (1.12)$$

as a matrix equation

$$Q^{\pi} = \mathcal{R} + \gamma \mathbf{P} \mathbf{\Pi}_{\pi} Q^{\pi} \quad (1.13)$$

where Q^{π} and $\mathcal{R}$ are vectors of size $|\mathcal{S}||\mathcal{A}|$, $\mathbf{P}$ is a stochastic matrix of size $|\mathcal{S}||\mathcal{A}| \times |\mathcal{S}|$, $P((s,a),s') = P(s,a,s')$, $\mathbf{\Pi}_{\pi}$ is a stochastic matrix of size $|\mathcal{S}| \times |\mathcal{S}||\mathcal{A}|$ that describes policy π: $\mathbf{\Pi}_{\pi}(s',(s',a')) = \pi(a';s')$. Or, writing it as a linear system, we have:

$$(\mathbf{I} - \gamma \mathbf{P} \mathbf{\Pi}_{\pi}) Q^{\pi} = \mathcal{R} \quad (1.14)$$

We can also write our approximation expression as a matrix equation. That is,

$$Q^{\pi}(s,a) \approx \hat{Q}^{\pi}(s,a;w) = \sum_{j=1}^{k} \phi_j(s,a) w_j \quad (1.15)$$

can be re-expressed in vector form. If we define a column vector:

$$\phi(s,a) = [\phi_1(s,a), \phi_2(s,a), \ldots, \phi_k(s,a)]^{\intercal} \quad (1.16)$$

and a matrix

$$\mathbf{\Phi} = \left[\phi(s_1,a_1), \ldots, \phi(s_{|\mathcal{S}|}, a_{|\mathcal{A}|});\right]^{\intercal} \quad (1.17)$$

we can compactly write $\hat{Q}^{\pi}$ as:

$$\hat{Q}^{\pi} = \mathbf{\Phi} \omega \quad (1.18)$$

In this framework it has been shown [23] that the error-minimizing (L2) approximation $\hat{Q}^{\pi}$ can be obtained by solving the following system for w:

$$\mathbf{A}(\pi)\omega = b \tag{1.19}$$

where:

$$\mathbf{A}(\pi) = \sum_{s\in\mathcal{S}}\sum_{a\in\mathcal{A}}\sum_{s'\in\mathcal{S}} P(s,a,s')q(s,a,s') \tag{1.20}$$

$$q(s,a,s') = \phi(s,a)(\phi(s,a) - \gamma\ \phi(s', \pi(s')))^{\intercal} \tag{1.21}$$

$$b = \sum_{s\in\mathcal{S}}\sum_{a\in\mathcal{A}}\sum_{s'\in\mathcal{S}} \mathcal{P}(s,a,s')\left[\phi(s,a)\mathcal{R}(s,a,s')\right] \tag{1.22}$$

If we are allowed to interact with the environment, we may be able to gather L experience tuples:

$$D = \{(s_i, a_i, r_i, s'_i)|\ i = 1, 2, \ldots, L\} \tag{1.23}$$

where r_i is the reward obtained when action a_i was taken in state s_i, resulting in a transition to state s'_i. If L of these experience tuples are collected, $\mathbf{A}$ and b can be approximated by [23]:

$$\tilde{\mathbf{A}}(\pi) = \frac{1}{L}\sum_{i=1}^{L}\left[\phi(s_i,a_i)(\phi(s_i,a_i) - \gamma\phi(s'_i, \pi(s'_i)))^{\intercal}\right] \tag{1.24}$$

$$\tilde{b} = \frac{1}{L} = \sum_{i=1}^{L}\left[\phi(s_i,a_i)r_i\right] \tag{1.25}$$

After solving for ω, one has immediately an improved policy π':

$$\pi'(s) = \arg\max_{a'} \hat{Q}(s, a') \tag{1.26}$$

which can then be used to generate a new $\mathbf{A}(\pi')$, re-solve for ω, and generate another improved policy π'', etc. This process is performed iteratively (essentially implementing an actor-critic policy iteration scheme in an approximated-Q environment) until the series of ω stabilizes, at which point one will have $\omega = \omega^*$ such that $\hat{Q}(\omega) = \hat{Q}^{\pi^*}$ [23]. Each iteration of the algorithm requires $O(L(k^2 + k|\mathcal{A}| + k^3))$ computational effort.

B. Reinforcement Learning in Multi-Agent Systems

Using reinforcement learning in the context of multi-agent systems presents several unique challenges. The first relates to how the multi-agent system is represented in the state-space of the MDP. A simple approach is to say that each agent has some action space $\mathcal{A}_i$ and that the joint action space of the multi-agent system is simply $\mathcal{A} = \mathcal{A}_i \otimes \mathcal{A}_2 \otimes \ldots \otimes \mathcal{A}_N$ where N is the number of agents in the system. As such, system actions can be represented by action vectors $\mathbf{a}$, where each component represents the actions taken by each system agent. In this case, the size of $\mathcal{A}$ is now exponential in the number of agents, $|\mathcal{A}| = |\mathcal{A}_i|^N$ (assuming the agents all have an action space with the same cardinality). This is unfortunate, since this means that $Q(s, a)$ must be defined over an exponential number of actions, which in turn means that most policy training algorithms, whose execution times are dependent at least linearly with the size of the action space will have computational requirements which scale exponentially with the number of system agents. Table I lists the computational

Table I. Computational complexity of common RL algorithms

Algorithm Algorithm	Computational complexity for single-agent system	Computational complexity on MAS domains
Value iteration (full backups)	$O(\vert S\vert^2\vert A\vert)$	$O(\vert S\vert^2\vert A\vert^N)$
Policy iteration	$O(\vert S\vert^2\vert A\vert + \vert S\vert^3)$	$O(\vert S\vert^2\vert A\vert^N + \vert S\vert^3)$
Q-learning	$O(\vert S\vert\vert A\vert^2)$	$O(\vert S\vert\vert A\vert^{2N})$
LSPI	$O(L(k^2 + k\vert\mathcal{A}\vert + k^3))$	$O(L(k^2 + k\vert\mathcal{A}\vert^N + k^3))$

complexity of several common reinforcement learning algorithms.

We will refer to this problem generally as the "exponential-action-space (EAS) problem", and the reader should note that it has implications for many aspects of reinforcement learning. For example, a subtle side-effect of the EAS problem occurs during action selection in multi-agent domains: since there are now an exponential number of available actions, an exponential number of $Q(s, a)$ values need to be evaluated before the joint action for all the system agents can be determined.

Despite these problems, using reinforcement learning in the multi-agent domain is appealing, and as such, some recent work has been done to extend reinforcement learning techniques such that they overcome EAS-type problems.

C. Related Research

There has already been a great deal of work done on how to solve the problem of how to efficiently train large, multi-agent systems. However, almost all of this research assumes some pre-knowledge of the agent system. Our position is differentiated because our methods require little to no system pre-knowledge before training. Some

of the extant approaches to training large systems are listed below.

1. Naive Single-Agent Training

As one approach to the problem of the computational complexity of RL, one may simply ignore the fact that system agents are part of a collective and train each agent independently of the others. This is certainly a computationally efficient approach (since training need only occur over a one-agent action space at a time), but this approach has significant drawbacks. At its core, the problem is this: from the perspective of each agent, the other system agents are part of the environment. Training the first agent n_1, the other agents are seen to execute whatever policy they executed when the training samples for n_1 were generated. Presumably however, when we train the next agent n_2, we will assign it a different policy than the one it followed when n_1 was being trained. Immediately, we have a conflict - since n_2 adopted a new policy, and n_2 was part of n_1's environment, n_1's environment has been altered. In fact, since n_1 finds itself in an effectively new environment (with n_2 adopting the post-training policy), the original policy of n_1 may perform sub-optimally.

Several authors have noted this problem. One particularly insightful description terms it "agents working at cross purposes" [53], and to some extent, if two agents to not cooperate at some level during learning, one risks not being able to learn a social-welfare optimizing policy [54]. Work has been done on methods to extend this kind of naive single agent training in a way that still preserves the reduction in computational complexity that goes with training each agent separately.

a. Relation to Game Theory

This notion of training at cross-purposes is slightly academic and difficult to get a feel for in the general case. However, there are specific, small-scale examples where this type of behavior becomes obvious. In [55] Hu and Wellman adapt Q-learning to a zero-sum, 2-player game. This toy-domain is useful because the authors are able to have each agent take into consideration both its own policy and the current policy of the other agent. This is accomplished by maintaining two separate Q tables, one for the agent itself, and one for the opposing agent. The authors show how this approach allows their system to converge to Nash equilibrium. Even with this sophisticated approach however, the author's raise the point that even when Nash equilibrium has been reached, there is no guarantee that the equilibrium is globally optimal.

b. Relation to Gradient Ascent

While training single agents at a time necessarily introduces the risk of training system agents to work at 'cross purposes' and never reaching a globally optimal system-wide policy, several authors have done work showing how, under the right conditions, one can train system agents one at a time and guarantee that the system will converge to a *Nash equilibrium.* A Nash equilibrium is a set of agent policies such that each agent cannot improve its local reward by changing its policy alone.

The basic approach involves thinking about reward as an energy surface over the space of agent policies. That is, imagine a multi-dimensional surface defined over the policies for each agent. Specifically, one might visualize a situation where each axis of the multi-dimensional energy surface corresponds to the policy of one of the system agents. This energy surface will have a global maximum, and more than likely

have many more local maxima. In this situation, the global maximum corresponds to the social-welfare maximizing policy set, and each local maxima represents a Nash equilibrium point.

If we were performing standard gradient ascent on such a surface, it would not be hard to reach one of these local maximum (although we couldn't guarantee that we would reach the global maximum). As such, several authors have shown how it is possible to perform gradient ascent in the policy space of an agent system. The central contribution from most of these works is a method to parameterize the policies such that they can be modified continuously. That is, gradient ascent requires that we be able to take infinitesimal steps, and much of the work in this area concerns how to parameterize policies so that this is possible, and proving convergence results when such infinitesimal updates are not possible because of the domain.

Bowling and Veloso [56, 57] have introduced a family of algorithms called WoLF (Win or Learn Fast) which provides a framework for parameterizing and updating agent policies such that convergence guarantees can be had. The name stems from the fact that the algorithm adjusts agent policies more quickly when performance is suboptimal (and hence the agent is more likely to be farther away from a Nash equilibrium) and more slowly when the agent is performing well (when it is most likely close to a Nash equilibrium).

In a related work, Kakade [58] showed that training a multi-agent system by iteratively performing approximate policy iteration agent-by-agent is mathematically *equivalent* to performing gradient ascent on a particular policy-quality measure (the long-term reward [58]). Approximate policy iteration is well suited to gradient-ascent-type learning since generally each policy is represented by a continuous-valued weight vector $\vec{\omega}$, and since policy iteration is generally performed in the space of $\vec{\omega}$, adjusting the policy weight vector directly provides a natural method to incrementally adjust

policies.

2. Coordinated Reinforcement Learning

Coordinated reinforcement learning is a framework for performing reinforcement learning (that is, it is used *with* an RL algorithm such as value iteration, policy iteration, etc.) which seeks to overcome the high computational costs of training policies over large numbers of agents [59, 60, 61, 62]. The technique starts out by assuming that the $Q(s,a)$ function for the multi-agent system can be written in the form

$$Q(\mathbf{s},\mathbf{a}) = \sum_{i=1}^{N} Q_{A_i}(\mathbf{s}_i,\mathbf{a}_i) \tag{1.27}$$

where $\mathbf{s}$ and $\mathbf{a}$ are vectors of states and actions available/performable by the *entire* multi-agent system, and $\mathbf{s}_i$ represents portion of the system state vector accessible to agent A_i and $\mathbf{a}_i$ represents the portion of the action vector performed by agent A_i.

Coordinated reinforcement learning (CRL) gets around the exponential action space problem because in many reinforcement learning algorithms (such as Q-learning and Least-Squares Policy Iteration), the single-agent Q functions can each be updated independently, as is shown in [59]. The method gets around the problem of multi-agent action selection by assuming that we possess a *coordination* graph of inter-agent dependencies. Groups of agents that are all interdependent must have their joint actions maximized over Q by brute-force search. That is, if agents A_1, A_2 and A_3 are all mutually dependent on each other, the maximizing multi-agent action $\mathbf{a} = \{a_{A_1}, a_{A_2}, a_{A_3}\}$ must be determined by testing all possible $Q(s,\mathbf{a})$ and choosing the collective action with the highest payoff. If, on the other hand, agent A_1 was dependent on no other agent, A_2 was dependent on A_1 only and A_3 was dependent

on A_3 only, then A_1's Q_1-function could be maximized first, and then, depending on which action was selected by A_1, agents A_2's action could be chosen to maximize $Q_2(s,a)$, etc. As [59] shows, this approach leads to system action selection which maximizes the total system $Q(s,a)$ function - again, as long as $Q(s,a)$ can be written as a linear sum of single-agent Q-functions. As such, coordinated reinforcement learning techniques are only appropriate when a lot of knowledge is possessed about the domain *a priori* - we must at least know the coordination graph for our agents and have enough knowledge of the domain to say whether a linear decomposition of the system-wide $Q(s,a)$ function would be appropriate.

3. Modified Reward Functions

Another approach is to modify the reward function each agent trains under in order to encourage each agent to not work at "cross purposes" with other system agents. Wolpert, et. al. [53] propose the use of a modified reward function, roughly of the form:

$$R_i(s,a) = R(s,a) - r(s,a) \tag{1.28}$$

where R_i is the reward function used for training agent n_i, R is the global reward function, and r is a created function that describes the reward the system would obtain if agent n_i "never existed". The motivation is to let n_i train under a reward function that isolates its singular contribution to global reward. The authors demonstrate the use of this class of functions in several experimental domains, though the reader should note that strictly, such an approach is contrary to the purposes of this present work. Here we seek to present methods for training multi-agent systems that do not

require foreknowledge about system structure. System knowledge is incorporated into the Wolpert approach through the creation of the reward function $r(s, a)$. In order to compute this function one must know how agent n_i interacts with other agents in order to create an MDP such that the actions of n_i do not affect the other system agents (again, trying to make n_i "never exist").

4. TPOT-RL

TPOT-RL (team-partitioned opaque-transition reinforcement learning) [11] is a reinforcement learning method specifically designed for training large teams of agents over very large state spaces. Most relevant here, the algorithm works by effectively reducing the state space over which training must occur by only training agents to act on states/actions in which they are capable of acting. The algorithm was initially developed for use in robotic soccer where not all agent actions are possible in every state, and some agents (because of their pre-assigned team position, etc), cannot act in certain states anyway. By using knowledge of the domain, this algorithm is able to shrink the state space the policy must operate on, and by doing so can effectively operate on extremely large state spaces. Other than that novelty, the algorithm acts much like Q-learning - letting agents determine from experience the environmental transition probabilities.

D. Extending Reinforcement Learning to Unconstrained Multi-Agent Domains

Here we will extend the work done on using reinforcement learning in multi-agent domains to the case of training multi-agent systems when *very little* is known about the

multi-agent environment. We call such domains 'unconstrained' multi-agent domains since pre-training, the learner has little information about the nature of the learning system (agent interactions, environment properties, etc.). This is in marked contrast to the domain for which much of the research cited above is intended.

We can encapsulate our research question for this dissertation as the following: How can reinforcement learning be scaled up to learn over *unconstrained* multi-agent domains?

In Chapter II, we introduce a method to reduce multi-agent training times by first partitioning agents into coalitions. Intra-agent coalitions are allowed to communicate and coordinate their actions while extra-coalitional agents are not. This is a fairly standard approach [63, 64, 64, 65, 66], and is also very computationally convenient because only intra-coalitional agents needs to be trained together. If k is the largest coalition size, then LSPI never needs to be performed over more than k agents at a time. This reduces the complexity of training from $O(C^N)$ to $O(C^k)$. The problem, however, is that of knowing which coalitions to form before training. Because of high training times, it is necessary to know which coalitions to form *before training occurs*. Coordinated Reinforcement Learning assumes that the system designer has enough pre-knowledge of the system to be able to manually form these clusterings pre-training.

We present an algorithm called Information-Based Coalition Formation (IBCF) that is able to form coalitions such that post-training the system performs well. The algorithm derives information-theoretic relationships between system agents solely from the training data (which is obviously available before training). We demonstrate our algorithm on several domains, and show that even though our algorithm requires much less pre-knowledge about our agent systems, it is able to perform competitively with CRL, which requires a great deal more knowledge for the algorithm to

execute successfully. Specifically, we demonstrate IBCF on several problem domains: the multi-agent inverted pendulum problem (where multiple agents with partial state information must coordinate to keep a pendulum upright,the SysAdmin problem domain [59], and a power-grid management domain [67].

In Chapter III we address the problem of the sample complexity of multi-agent reinforcement learning. The sample complexity of a learning process is the number of samples that a learner must be exposed to in order to, with a high probability, learn a given concept. After reviewing how the sample complexity for multi-agent RL grows exponentially with the number of system agents, we introduce a framework to perform 'hybrid' reinforcement learning, where the learner uses not only training examples (of which there may not be enough because of the number of agents in a system), but also user-provided domain knowledge. In the context of this dissertation, we assume that whatever domain knowledge we *do* have may be imperfect and potentially be in the form of fuzzy-logic based rules. We develop a mechanism to integrate this fuzzy domain knowledge into the reinforcement learning process and demonstrate how the sample complexity of a RL problem can be reduced through our integration mechanism operating on a simple pole-balancing controller.

In Chapter IV we overcome a significant problem which hinders the use of continuous action spaces with the LSPI algorithm on some domains. Continuous action spaces are often required for systems dealing with analog control. If one does not use a continuous action spaces in such domains, one is forced to finely partition up the action space, resulting in increased computational training complexity, and compounding the effects of exponential multi-agent RL complexity scaling. Inherently LSPI is able to learn over continuous action spaces, though we present some subtle problems that manifest when continuous LSPI is used on 'narrow-Q' domains, or domains where, because of a high discounting factor γ or extremely delayed rewards, the

Q values for most states/actions are very similar. We demonstrate how continuous action select fails on narrow-Q domains and present a novel heuristic for repairing the narrow Q values. After discussing the potential non-optimality introduced by the heuristic, we demonstrate the successful use of the heuristic on a windy, inverted pendulum domain. In this domain the controller must be able to apply strong corrective forces when the pendulum experiences strong 'gusts' of wind, and only apply soft corrective forces otherwise. We demonstrate how, using our normalization technique, we are able to train a continuous-action controller over a domain which otherwise required a segmented action space.

Finally, in Chapter V, we present an implementation of the LSPI algorithm which runs almost exclusively on the GPU, and analyze the performance enhancements that come from exploiting the GPU's SIMD architecture. Specifically, we concentrate on the part of the LSPI algorithm involved in constructing the $\mathbf{A}$ matrix, and show how by parallelizing this construction over experience tuples, we can achieve over a $4.5x$ speedup using standard CPU-only hardware, which is close to the largest theoretical speedup via parallelization on our test hardware. Finally, we discuss some limitations of the hardware-induced limitations of the implementation, and provide a Brook-based source code listing of our GPU LSPI implementation.

CHAPTER II

REDUCING REINFORCEMENT-LEARNING COMPLEXITY THROUGH COALITION FORMATION

A. Introduction

In this chapter we examine how a performance gain in multi-agent reinforcement learning can be realized by first partitioning system agents into a set of *coalitions*. The motivation is a simple one and has been repeated throughout this dissertation: the computational requirements for training policies over multiple agents scales exponentially with the number of agents. The reason is that the action-space of a multi-agent system scales exponentially with the number of agents, and most reinforcement learning algorithms scale at least linearly with the size of the training action space.

Thus we are faced with two competing objectives. One the one hand, computational considerations urge training as few agents together as possible. On the other hand, performing reinforcement learning on large numbers of agents simultaneously generates policies which maximize reward over joint agent actions, leading to phenomena such as agent 'coordination' and 'cooperation'.

We seek to find a balance between these two objectives - we will present a method which trains agents together where such training results in significantly higher agent performance, but otherwise trains agents separately to keep computational requirements to a minimum. Specifically, we will present a method which first groups agents together into cooperative *coalitions* and then attempts to generate policies which maximize system reward over the joint actions of each agent coalition.

Such an approach is inherently an approximation in the sense that because we separate our agents into coalitions and deprive them of being able to coordinate their actions, we may not be able to achieve the globally optimal policy. In this spirit of this work however, we show how the errors introduced by this approximation are bounded and demonstrate the effective use of this technique on several difficult experimental domains.

Generally, the process of coalition formation, where distinct autonomous agents come together to act as a coherent group is an important form of interaction in multi-agent systems. In many domains, partitioning a collection of agents into judiciously chosen coalitions can result in significant performance benefits to the entire agent system such as reduced agent coordination costs and training times, and increased access to pooled resources. The use of coalitions has been advocated in a wide variety of domains including electronic commerce [68] (purchasing agents pooling their requirements to obtain larger discounts), grid-computing [69], and e-business [70] (where agents can come together to form organizations to fill market niches). The coalition formation process can be viewed as being composed of three main activities [66]:

1. *Coalition Structure (CS) Generation*: The creation of coalitions such that agents in a given coalition coordinate their actions while agents in different coalitions do not. This involves partitioning the agents into disjoint sets, and such a partition is called a *coalition structure*. For example, in a multi-agent system composed of three agents n_1, n_2, n_3, there are seven possible coalitions: $\{n_1\}$,$\{n_2\}$,$\{n_3\}$, $\{n_1, n_2\}$, $\{n_2, n_3\}$, $\{n_3, n_1\}$, $\{n_1, n_2, n_3\}$. A valid coalition structure would be $\mathcal{CS} = \{\{n_1\}, \{n_2, n_3\}\}$

2. *Coalition Value Maximization*: In this step, the resources of the individual coalition agents are pooled to maximize the payoff to each coalition. In a market-based environment, this might mean literally combining financial resources to gain

more bargaining sway.

3. *Payoff Distribution*: Since coalition formation generally occurs in the context of autonomous, self-interested agents, at some point reward distributed to a coalition must be parceled to its member agents, and there are various common methods for doing this - equally among members, proportional to the agent's contribution to the coalition, etc.

Recent work on coalition formation has concentrated on the first of these steps (CS generation), and we will continue that tradition here. Generally, CS generation takes place in the context of some valuation function $V(\mathcal{CS})$ which measures the fitness of a given coalition structure. If the task structure of the environment is known ahead of time, this function may be directly evaluatable as in [64], while in other contexts, one may use the sum of individual agent reward payoffs (which would require performing all three steps of the coalition formation process).

In general, the goal of CS-generation is to construct a coalition structure which maximizes this valuation function $V(\mathcal{CS})$. Many authors have proposed algorithms for coalition formation in different situations, such as when optimality bounds are required [66, 71], or when coalition size can be bounded from above [64]. While much of this work has centered on deterministic algorithms, the field has seen some application of stochastic methods as well (particularly genetic algorithms [65]).

Traditionally, CS-generation requires access to the valuation function $V(\mathcal{CS})$ (for example, in the Sen & Dutta work, $V(\mathcal{CS})$ is needed to perform fitness evaluation in the genetic algorithm). And in fact, many accesses to $V(\mathcal{CS})$ may be required before a CS generation algorithm terminates. If the nature of the agent system, its environment, the task structure, etc. are known *a priori*, evaluating this function may be able to be done directly (as in [64]). However, in other cases, such as when dealing with coalition formation among physical robotic systems, valuing a coalition

structure may need to be done empirically by retraining the system to respect the CS and then letting the system interact with the environment, which might be an expensive undertaking.

B. Contribution

In this chapter work we look at the problem of coalition structure generation when repeated evaluation of $V(\mathcal{CS})$ is not feasible. This is at heart a functional maximization problem, made more complex since direct evaluation of the objective function is not possible. Fortunately, similar problems have been solved in related fields.[72, 73, 74, 75]

1. Classifier Training and Feature Selection

A similar problem is encountered by the classifier-training community with the problem of features selection. Feature selection is of great interest to the classifier training community because training classifiers over large numbers of features can incur significant computational costs, as well as invite potential problems with overfitting [76, 77].

One solution is to applying feature selection on a training corpus before using it to train a classifier can reduce the dimensionality of a training corpus by discarding all but the features necessary for the classifier to function well. This training corpus is used to train a classifier, it is desirable to keep features which make the error rate of the final trained classifier is as low as possible.

There are two approaches to solving this problem. The first is a family of algo-

rithms known as “wrappers” [74] and the second is a family of algorithms know as “filters”.

a. Wrappers

The simplest wrapper algorithms operate by adding features in a greedy fashion until a [74] maximum classifier performance is obtained. A set of features $\mathcal{F}$ is maintained, and at each iteration of the wrapper-generation algorithm a new 'test' feature is added to $\mathcal{F}$. The classifier is then trained over $\mathcal{F}$ and its post-training performance evaluated. For a set of N features, this approach requires up to $N(N-1)/2$ iterations, and consequently requires $N(N-1)/2$ calls to the classifier training algorithm. Although effective, if the time required to train the classifier is high, and, more seriously if the number of *features* is high, this approach can be unacceptable because of computational limitations. Additionally, because the algorithm is greedy, optimality cannot be guaranteed. Some work has been done on look-ahead extensions to this simple greedy technique to ameliorate that problem [78].

b. Filters

A second approach is to use “filters”. This approach constructs a suitable feature-set *before* training the classifier or evaluating its performance. Because feature selection occurs before classifier training, the performance of the classifier cannot be used to determine which features will result in a low-error post training classifier. Instead, intrinsic properties and interactions of the features themselves must be used to determine which set of features will yield a high performance classifier.

Although there are many diverse algorithms available for filter-based feature

selection, in this chapter we will concern ourselves with the class of *information-theory filters*. Information-theory based filters use statistical properties of the training corpus to determine which sets of features will result in the lowest error for the post-training classifier. As a generalization, most such approaches seek to select those features which possess the most statistical information about class labels.

As a side-note, occasionally research on information-based filters is performed under the name of 'minimum entropy filter' research. Mathematically, however, entropy and information are identical aside from a sign, and the difference is more one of perspective and terminology [73, 79].

A wide variation of information-based filtering techniques exist, perhaps the largest distinguishing feature between the approaches being the degree to which "inter-feature-interactions" are considered. Some simple information-based filters merely choose the n features in a greedy fashion that convey the highest amount of information about class labels (as in [80]). This approach does not consider inter-feature interactions, and can result in many redundant features. More sophisticated information-based filters look at binary interactions between features (examining the mutual information between features) [72, 75], and still other algorithms consider full n^2-type interactions between features.

2. Approach

The parallels between filter methods in classifier feature selection and our situation are significant. In both cases there is a need to maximize an objective function. In the case of classifier training this is the error $\epsilon(C)$, here it is the coalition-structure valuation function $V(\mathcal{CS})$.

As such, we will view coalition formation from an information theoretic stand-

point. First we will describe an abstract coalition structure CS_0 such that each coalition $C_i \in CS_0$ maximizes certain statistical properties (specifically the Keibler-Leibeck measure over states, actions, and reward). We will show how by maximizing this KL-measure across its component coalitions CS_0 also maximizes policy quality across the system. We will then propose an algorithm to go about forming coalition to approximate CS_0. Since our algorithm will inherently only approximate this coalition structure CS_0, we will finally discuss how policy quality varies with approximation errors.

We will need a pre-existing training corpus to calculate the KL measure over coalitions, and as such, here we will concentrate exclusively on corpus-based reinforcement learning. Specifically we will use the Least-Squares Policy Iteration [23] algorithm to train, where experience data is collected in one stage and training occurs afterward).

C. Preliminaries

1. Coalitions and Spaces

Definition: Coalition. A *coalition* of $C = \{n_i, \ldots, n_k\}$ groups a set of agents together such that they are able to act in a coordinated fashion. At each time-step each agent possesses a state $s^{n_i} \in \mathcal{S}_{n_i}$ and performs an action $a^{n_i} \in \mathcal{A}_{n_i}$ at each time-step. We will treat the coalition as a composite-entity with a state space $\mathcal{S}_C = \bigotimes_{n_i \in \mathcal{C}} \mathcal{S}_{n_i}$ and an action space $\mathcal{A}_C = \bigotimes_{n_i \in \mathcal{C}} \mathcal{A}_{n_i}$. At each time-step the coalition has a state $s^C = \bigcup_{n_i \in \mathcal{S}} s^{n_i}$ and performs an action $a^C = \bigcup_{n_i \in \mathcal{A}} a^{n_i}$.

■

Implicit in this definition is that an agent n_h in a coalition C_i has no access to the state or action of another agent n_j in coalition C_k.

Definition: Coalition Structure. A *coalition structure* $\mathcal{CS} = \{C_1, \ldots, C_{|\mathcal{CS}|}\}$ is a set of coalitions $\mathcal{C}_i$ such that each system agents n_i is a member of precisely one coalition $C_j \in \mathcal{CS}$. The state-space of a coalition structure is $\mathcal{S}_C = \bigotimes_{C_i} \mathcal{S}_{C_i}$ and the action space is $\mathcal{A}_C = \bigotimes_{C_i} \mathcal{A}_{C_i}$. At each time-step a coalition-structure has a joint state $s^{CS} = \bigcup_{C_i \in \mathcal{CS}} s^{C_i}$ and performs a joint action $a^{CS} = \bigcup_{C_i \in \mathcal{CS}} a^{C_i}$.
■

Most generally, coalitions serve only to *partition* agents into groups such that each agent may belong to more than one coalition [81]. Here however, because an agent can have only a single policy, and we will eventually train one policy per coalition, we will demand that an agent belong to only one such coalition in a coalition-structure.

2. Reduced Training Corpi

Before policy training can occur, LSPI requires that we collect a number of *experience tuples* (collectively referred to as a training corpus $\mathcal{T}$) by allowing our agent system to interact with the environment:

$$\mathcal{T} = \{\langle \mathbf{s_t}, \mathbf{a_t}, \mathbf{s'_t}, r_t \rangle\} \tag{2.1}$$

Not all of this information is available to each agent. In accordance with our previously-introduced notation, we will write the training corpus available to coalition C as a

reduced corpus T^C

$$\mathcal{T}^C = \{\langle s_t^C, a_t^C, s_t'^C, r_t \rangle\} \tag{2.2}$$

3. Policies

Definition: Single Agent Policy. A *single-agent policy* π^{n_i} maps the state of an agent n_i, s^{n_i}, to the action of that agent n_i, a^{n_i} such that $a^{n_i} = \pi^C(s^{n_i})$.

■

Definition: Coalition Policy. A *coalition policy* π^C maps the joint state of a coalition C, $s^{\mathcal{C}}$, to a joint action of coalition C, $a^{\mathcal{S}}$ such that $a^{\mathcal{S}} = \pi^C(s^{\mathcal{C}})$.

■

Definition: Coalition Structure Policy. A *coalition-structure policy* π^{CS} maps the joint state of a coalition structure CS, $s^{\mathcal{CS}}$, to a joint action of coalition structure CS, $a^{\mathcal{SS}}$ such that $a^{\mathcal{CS}} = \pi^{CS}(s^{\mathcal{CS}}) = \bigcup_{C_i \in \mathcal{CS}} \pi^{C_i}(s^{C_i})$.

■

4. Coalition Structure Valuation

Definition: Value of a Coalition Structure. The value of a coalition structure $\mathcal{CS} = \{C_i\}$ will be determined by training and evaluating $|\mathcal{CS}|$ policies π^{C_i}, one policy to deterministically control the agents in each coalition $C_i \in \mathcal{CS}$. The value of a coalition structure $V(\mathcal{CS})$ is:

$$V(\mathcal{CS}) = \rho(\pi^{CS}) = \rho(\{\pi^{C_i}\}) = E\Big[\sum_{t=1}^{\infty}\gamma^{t-1}r_t|\mathbf{s_0}, \{\pi^{C_i}\}\Big] \tag{2.3}$$

with respect to some arbitrary initial state $\mathbf{s_i}$, where each π^{C_i} has been trained via reinforcement learning, and where r_t is the global reward of all agents in all coalitions.

5. Choice of Reward Function

Notice that our policy quality is defined in terms of the global system reward. This is in contrast to other related work where different reward functions are used for each agent, etc (see [53] for an example of this). Specifically, in the Wolpert, et. al. work the authors delve into the problem of how to train agents independently so that they do not "work at cross-purposes" [53]. The authors show results that using a modified reward function for each agent allows them to more or less accomplish this.

Our work is related to this work in that both purpose to develop a method for training large multi-agent systems that does not have exponential computational requirements. However, the Wolpert work, like CRL, requires foreknowledge of the agent system. Specifically, in order to construct the modified reward function for agent n_i, a modified MDP must be constructed where agent n_i's actions do not affect the other agents (or in Wolpert's terminology, the MDP must be constructed as if n_i "had never existed"). This requires knowledge of how n_i interacts with other system agents. If one has this information, the author's show that training can be performed agent-by-agent with reasonable results.

We treat the domain where little or no a priori information is available about how the system agents relate to each other. We seek to generate adequate coalitions purely from training data, without the need to probe the system further.

Thus, we are more or less restricted to using the global reward function. The purpose is to develop a coalition formation algorithm that operates with minimal information about system structure. Using a different reward function for an agent or group of agents would seem to require additional, agent-specific knowledge and would more or less violate the spirit of this work.

D. Information Theory

1. Introduction

Information theory is a mathematical theory of data transmission and storage. Founded by Claude Shannon in 1948, [? 82], information theory was originally designed to describe the amount of information that could be successfully transmitted over a noisy communication channel (such as a telephone line). The theory is far more general however, and can be used to describe the amount of information contained in very arbitrary random variables.

Specifically, Shannon defined the amount of information contained in a random variable X with probability distribution $p(X)$ to be:

$$I_X = -\int_{-\infty}^{+\infty} p(x) \log p(x) dx \tag{2.4}$$

Notice that this quantity is maximized when $p(x)$ is uniform over the probability domain. As such, information can be thought of as measuring the *uncertainty* inherent in a probability distribution. That is, for probability distributions where most of the probability is concentrated in a very small portion of the domain, even when we find out the exact value of x, we have really not gained that much *information* since we

were already fairly certain of the value x beforehand. On the other hand, imagine that $p(x)$ is very spread out such that beforehand we have little idea of the exact value of x. In this case, upon determining the exact value of x, we gain a *lot* of information since we were initially extremely uncertain.

The notion of information can be extended to multiple variables in a fairly straightforward manner, and one can speak of the information contained between *pairs* of random variables. The appropriate metric for this is *mutual information*, and is conceptually similar to the correlation between variables. However, mutual information is specifically tied to the amount of *information* one random variables tells about about another random variable.

For example, imagine that we have a situation with two random variables X and Y. If we want to choose a biometrics domain, we might imagine that X represents the gender of a person, and Y represents their height. Before knowing either X or Y, we'll say that X is $P(X = male) = 0.5$, $P(X = female) = 0.5$, and that Y is a Gaussian distributed around about 5'5".

In general, if we have two random variables X and Y, then the mutual information between the two variables is:

$$M(X;Y) = \int_X \int_Y p(x,y) \log \frac{p(x,y)}{p(x)p(y)} dxdy \tag{2.5}$$

However, imagine that for a given subject, we are able to determine X. Immediately, the distribution of Y changes. Let us say that we learn that the subject is male - in this case the probability distribution for Y changes. Since the subject is male, Y's mean would shift to larger height values and the variance would shrink as well. Likewise, if we were able to figure our Y for a subject (lets say we found out that $Y = 6'8"$), the probability distribution for X would change as well. In this case,

since we know that the subject is very tall and is therefore most likely male, the conditional probability distribution for X might look like $P(X = male|Y) = 0.9$, $P(X = female|Y) = 0.1$.

In this example the random variables X and Y were not independent, and knowing one value told us something about the other value. We could look at this in a slightly different way: knowing the value of one of the variables *decreased* the amount of information that was available in the other one. That is, in the height example, once we had figured out that the subject was male, we were more certain about the subject's height (the variance of the distribution decreased) ... and thus, if we were to measure the subject's height, we would find out less information.

2. Application to Reinforcement Learning

We can write down the mutual information in the training corpus relating states and actions to reward as the following:

$$I(r; \mathbf{s}, \mathbf{a}) = \sum_{\mathbf{a}} \int_r \int_{\mathbf{s}} P(\mathbf{s}, \mathbf{a}, r) \log \frac{P(\mathbf{s}, \mathbf{a}, r)}{P(\mathbf{s}, \mathbf{a})P(r)} \tag{2.6}$$

The reader may worry about the raw estimation of $P(\mathbf{s}, \mathbf{a}, \mathbf{s}', r)$ that needs to occur in order for the expression to be evaluated. We will discuss in a few sections how to go about practically computing this integral. We can also speak of the information available to agents in a coalition C_i:

$$I(C_i) \equiv I(r; \mathbf{s}^{C_i}, \mathbf{a}^{C_i}) \tag{2.7}$$

$$= \int_r \int_{\mathbf{s}^{C_i}} \int_{\mathbf{a}^{C_i}} P(\mathbf{s}^{C_i}, \mathbf{a}^{C_i}, r) \log \frac{P(\mathbf{s}^{C_i}, \mathbf{a}^{C_i}, r)}{P(\mathbf{s}^{C_i}, \mathbf{a}^{C_i})P(r)} \tag{2.8}$$

Since our goal is to maximize the amount of information possessed by each coalition, we will attempt to form coalitions to maximize this quantity. Intuitively this makes sense - if each coalition has a great deal of information about how states and actions relate to reward, reinforcement learning should be able to take this information and produce a high-performance policy. However, there are also very good quantifiable reasons for forming coalitions for maximize this quantity as well as we will discuss shortly. However, before we proceed, the following observation will be useful to us in our analysis.

Observation: If C is a coalition, and T^C is a training corpus generated for use by the LSPI algorithm, $P(s^C, a^C) = \prod_{n_i \in C} P(s^{n_i})P(a^{n_i})$ when these probabilities are calculated over T^C because we know *a priori*, these probabilities are independent to begin with.

Reason: This theorem does not claim to prove anything new about the above-mentioned probabilities, only to make rigorous a probabalistic relationship that is *required* to be true by the LSPI algorithm. That is, this is not making a claim about $P(s^C, a^C)$ being estimated by $\prod_{n_i \in C} P(s^{n_i})P(a^{n_i})$, instead, this theorem points out that the training samples in LSPI must be collected in such a way to satisfy this requirement for the LSPI algorithm to work (this is given as one of the requirements of the algorithm in the presentation papers [23]. Specifically, this algorithm *requires* that the training examples randomly sample the state and action spaces of all agents to be trained before training begins. This requirement is equivalent to the above

probability relationship.

In practice this sampling is done by seeding the agent system to a random state and then having the system perform the random policy and measuring reward, ensuring that each training sample is taken from a random point in state/action space. Since we will only use the LSPI learning algorithm here, and we must have this even sampling to satisfy the constraints of the algorithm, the observation must hold.

■

E. Why Maximize Mutual Information?

Mutual information over a set of variables can be written in terms of the Leibler-Kullback divergence (KL measure). This measure is written $D(p||q)$ where p and q are two probability distributions, and D measures the distance between the two distributions (although D is not technically a distance metric because it is not symmetric and it does not obey the triangle rule).

$$D(p||q) = \int_X p(x) \log \frac{p(x)}{q(x)} dx \tag{2.9}$$

Now let X and Y be random variables with a joint distribution $p(X, Y)$ and marginal distributions $p(x)$, and $p(y)$. We can write the mutual information $I(X;Y)$ in terms of the KL measure:

$$I(X;Y) = \int_X \int_Y p(x,y) \log \frac{p(x,y)}{p(x)p(y)} dxdy = D(p(x,y)||p(x)p(y)) \tag{2.10}$$

and in this context, where we talk about the information contained in coalitions, we

can write:

$$I(C) = I(s^C, a^C; r) \equiv D(C) = D(P(s^C, a^C, r)||P(s^C, a^C)P(r)) \tag{2.11}$$

It is interesting to express the coalition information in terms of the KL measure because many theorems about the KL measure have been proven, and we can use this foundation to prove some theorems relevant to the coalition-information system we consider here.

Theorem 1: The system policy quality $\rho(\pi^C)$ is maximized when the KL measure $D(C) = D(P(s^C, a^C, r)||P(s^C, a^C)P(r))$ is maximized with respect to single-agent membership additions to some coalition C.

Proof: Consider the case of a coalition C with KL measure $D(C)$. Now let us add an agent n_i to the coalition C to form coalition C'. We will use the property of the KL measure that, for three distributions $p(x), q(x), r(x)$, we have:

$$D(p(x)||q(x)) = D(r(x)||q(x)) \ \ iff \ \ p(x) = r(x) \tag{2.12}$$

such that if:

$$D(P(s^C, a^C, r)||P(s^C, a^C)P(r)) = D(P(s^{C'}, a^{C'}, r)||P(s^{C'}, a^{C'})P(r)) \tag{2.13}$$

and we rewrite

$$D(P(s^{C'}, a^{C'}, r)||P(s^{C'}, a^{C'})P(r)) = D(P(s^{C'}, a^{C'}, r)||P(s^{C}, a^{C})P(s^{n_i}, a^{n_i})P(r)) \tag{2.14}$$

since $P(s^C, a^C)$ (for any coalition) is by design just a product individual state/action probabilities, and we also rewrite

$$D(P(s^C, a^C, r)||P(s^C, a^C)P(r)) = A \tag{2.15}$$

$$A = D(P(s^C, a^C, r)P(s^{n_i}, a^{n_i})||P(s^C, a^C)P(s^{n_i}, a^{n_i})P(r)) \tag{2.16}$$

which we can do because:

$$D(p(x)||q(x)) = D(p(x)r(x)||q(x)r(x)) \tag{2.17}$$

and, we have that:

$$D_A = D_B \tag{2.18}$$

where

$$D_A = D(P(s^C, a^C, r)P(s^{n_i}, a^{n_i})||P(s^C, a^C)P(s^{n_i}, a^{n_i})P(r)) \tag{2.19}$$

$$D_B = D(P(s^{C'}, a^{C'}, r)||P(s^C, a^C)P(s^{n_i}, a^{n_i})P(r)) \tag{2.20}$$

such that we have finally:

$$P(s^{C'}, a^{C'}, r) = P(s^C, a^C, r)P(s^{n_i}, a^{n_i}) \tag{2.21}$$

$$P(r|s^{C'}, a^{C'}) = \frac{P(s^{C'}, a^{C'}, r)}{P(s^{C'}, a^{C'})} \tag{2.22}$$

$$P(r|s^{C'}, a^{C'}) = \frac{P(s^{C'}, a^{C'}, r)}{P(s^{C}, a^{C})P(s^{n_i})P(a^{n_i})} \tag{2.23}$$

and by the previous result

$$P(r|s^{C'}, a^{C'}) = \frac{P(s^{C}, a^{C}, r)P(s^{n_i}, a^{n_i})}{P(s^{C}, a^{C})P(s^{n_i})P(a^{n_i})} \tag{2.24}$$

$$P(r|s^{C'}, a^{C'}) = P(r|s^{C}, a^{C}) \tag{2.25}$$

Such that adding agent n_i to the coalition has absolutely no effect on our the reward function and therefore has no effect on the final policy. Also, note that this result holds even if we add other agents to the coalition *before* adding agent n_i. That is, by the properties of the KL measure we have that if:

$$D(P(s^C, a^C, s^{n_i}, a^{n_i}, r)||P(s^C, a^C, s^{n_i}, a^{n_i})P(r)) = D(P(s^C, a^C, r)||P(s^C, a^C)P(r)) \tag{2.26}$$

then

$$D(P(s^C, a^C, x, s^{n_i}, a^{n_i}, r)||P(s^C, a^C, x, s^{n_i}, a^{n_i})P(r)) = A \tag{2.27}$$

$$A = D(P(s^C, a^C, x, r)||P(s^C, a^C, x)P(r)) \tag{2.28}$$

for any variables x. This means that when a coalition has a maximized KL measure

for single-agent additions, then the KL measure is maximized even under arbitrary multi-agent additions.

Intuitively this make sense - adding an agent n_i into a coalition should increase the information contained in that coalition. However, if we add another agent n_j in first before we add in n_i, we can never get *more* information out of n_i. In fact, n_j may provide some of the information that n_i would provide, such that the incremental information gain provided by n_i can only be less if other agents are added first.

■

Theorem 2: Coalition policy quality can never increase by removing an agent from a coalition.

Proof: Let $\mathcal{C}_1$ be a coalition, and let $\mathcal{C}_2$ be that coalition with agent n_i removed. The policy π^{C_1} maps states to actions as:

$$\pi^{C_1} : \{s^{C_2}, s^{n_i}\} \mapsto \{a^{C_1}, a^{n_i}\} \tag{2.29}$$

and policy π^{C_2} maps:

$$\pi^{C_2} : s^{C_1} \mapsto a^{C_1} \tag{2.30}$$

Thus, the entirety of the space of policies π^{C_2} is contained in the space of policies π^{C_1}, and no *better* policy can be had by removing agent n_i and adopting a policy π^{C_1}.

■

Corollary: When mutual information is maximized, coalition policy quality cannot increase by any membership changes.

Proof: By **Theorem 1** we have that under these conditions policy quality cannot increase through agent additions, and by **Theorem 2** we have that policy quality cannot increase though agent subtractions. Therefore, policy quality cannot increase under arbitrary membership changes.

■

1. IBCF Algorithm

We will now introduce an algorithm which will seek to form coalitions to maximize mutual information. According to the just-derived theorems, we have reason to believe that if we could maximize each coalition's mutual information (or could construct each coalition so that additional agents did not increase the coalition's information), this would result in high-quality coalitions post-training. As such, our algorithm, which we will term IBCF (Information-Based Coalition Formation), will seek to greedily maximize coalition information, stopping only when no more information gain is possible with a given coalition or when a coalition has reached the maximum allowable size. The code for the IBCF algorithm is listed in Appendix A.

F. Analysis of Computational Requirements

1. Introduction

Conceptually this algorithm works by greedily attempting to form coalitions with maximum information content. It does this iteratively. Starting with a given coali-

tion C, the algorithm blindly tried adding different agents, and chooses to add the agent which most dramatically increases the coalition's information. If the maximum coalition size is reached, the coalition is 'closed', a new coalition is created, and the process continues.

Since, at each iteration, one agent is assigned to a some coalition, the overall cost of the algorithm will scale as $O(N^2)$. That is, in the first pass, all N agents are added to an empty coalition to see which one possesses the most information, on the next pass, the $N-1$ remaining agents are each added to the 1-agent coalition, etc.

2. Analysis

Result: The ICBF algorithm runs in $O(N^2 2^K L)$ where N is the number of system agents, K is the largest allowed coalition size and L is the number of training examples used.

Derivation: In the inner-most loop, the information calculation integral takes $O(2^{|C|}|L|)$ work, where $|C|$ is the current coalition size and $|L|$ is the number of training examples. Simply, this is because to compute the information integral one can always grid up the integration space into finite-size cells, of which there will be $O(2^{|C|})$ of them (two dimensions for each coalition agent, one state, and one action dimension). Naively, one could simply iterate through the $O(2^{|C|})$ cells, and, for each one, sort through each of the $|L|$ training tuples to see whether each tuple occupied the cell. If the maximum coalition size is K, then this cost never reaches above $O(2^K|L|)$.

Since each time the inner loop is run an agent is removed from the available POOL agents, the inner information integral is computed N times the first iteration, $N-1$ times the second iteration, etc. Thus, we have that the total execution time

for the algorithm is:

$$O(N^2 2^K |L|) \tag{2.31}$$

where N is the total number of system agents, and K is the largest permissible coalition size.

■

3. Discussion

This algorithm still scales exponentially with the number of agents. In this respect we have not gained a strict scaling advantage, though we are able to specify a fixed maximum coalition size K to help keep computational requirements reasonable. In this respect, this algorithm is much like the coalition formation algorithm presented in [**?**], since both avoid exponential requirement scaling by capping the largest coalition size.

Additionally, note that this does not include estimates for training time. Implicitly training time is bounded by the size of the generated coalitions. As we have pointed out repeatedly in this work, the computational requirements for most reinforcement learning algorithms scales as $O(C^N)$ where N is the number of agents trained simultaneously. As such, even after running the ICBF algorithm, an additional training step must be performed which will incur an $O(C^K)$ cost. In some ways this additional cost does not change anything since the overall scaling for even the combined coalition formation / training procedure is still $O(N^2 2^K |L|)$.

G. Errors and the Information Integral

We will now discuss some issues related to the numerical accuracy of the information integral calculation. Obviously, the probabilities in the integral are continuous and cannot be evaluated via sampling at every point in the state/action domain. This can be done by standard Newton-Cotes-type numerical integration, and we will use this framework to provide a rough bound on how errors in this integral depend on system parameters.

Theorem 3: If the information integral is calculated for a m-agent coalition via first-order Newton-Cotes numerical integration, the total error of the integral is bounded by:

$$\epsilon_{total} < \left[\frac{h^3}{24} |f''(\xi)| + O\Big(\sqrt{\frac{1}{4n}} \log \left[1 + \sqrt{\frac{1-h^k}{nh^k}} \right] \Big) \right] \tag{2.32}$$

where f is the coalition information function, and as usual, ξ is chosen to maximize the derivative, $k = 2m+1$ is the number of integration variables, n is the number of experience tuples used for training, and h is the grid-size used for the numerical integration.

Proof: If we are calculating the information integral for a m-agent coalition, we will be integrating over $k = 2m+1$ variables (a state/action pair for each agent, plus global reward), and for the sake of discussion we will assume that each integration variable assumes the range $[0, 1]$. Our information integral will be of the following form:

$$\hat{I}(s^C, a^C; r) = \int_{x_1} \int_{x_2} \dots \int_{x_k} \hat{P}_1 \log \frac{\hat{P}_2}{\hat{P}_3} \tag{2.33}$$

where P_1, P_2, P_3 are the probability distributions from the mutual information expression. Assume that each of these probability distributions has some error caused by sampling the probability at a given point by a finite number of samples:

$$P_i = \hat{P}_i + \epsilon_i \tag{2.34}$$

Such that:

$$\hat{I}(s^C, a^C; r) = \int (P_1 + \epsilon_1) \log \left[\frac{P_2 + \epsilon_2}{P_3 + \epsilon_3} \right] \tag{2.35}$$

$$\hat{I}(s^C, a^C; r) = \int (P_1) \log \left[\frac{P_2}{P_3} \right] + E_1 + E_2 + E_3 + E_4 \tag{2.36}$$

$$E_1 = \int (P_1 + \epsilon_1) \log \left[1 + \epsilon_2 / P_2\right] \tag{2.37}$$

$$E_2 = - \int (P_1 + \epsilon_1) \log \left[1 + \epsilon_3 / P_3\right] \tag{2.38}$$

$$E_3 = \epsilon_1 \int \log P_2 \tag{2.39}$$

$$E_4 = -\epsilon_1 \int \log P_3 \tag{2.40}$$

such that:

$$\hat{I}(s^C, a^C; r) = \int \left[(P_1) \log \left[\frac{P_2}{P_3} \right] \right] + \int O(\epsilon \log \left[1 + \epsilon'\right]) \tag{2.41}$$

and since our integration domain is a unit hypercube

$$\hat{I}(s^C, a^C; r) = \int \left[(P_1) \log \left[\frac{P_2}{P_3} \right] \right] + O(|\epsilon \log [1 + \epsilon'] |) \tag{2.42}$$

Now, the probability errors will decrease as the law of large number such that:

$$\epsilon = \sqrt{\frac{p(1-p)}{n}} \tag{2.43}$$

$$\epsilon' = \frac{1}{p} \frac{\sqrt{p(1-p)}}{\sqrt{n}} = \sqrt{\frac{1-p}{np}} \tag{2.44}$$

where we have abused notation slightly - p is in fact P_1 or P_2 or P_3 depending on which error term is under discussion, and where n is the number of samples used to estimate the probabilities. We also have (maximizing the above equations)

$$|\epsilon| < \sqrt{\frac{1}{4n}} \tag{2.45}$$

$$|\epsilon'| < \sqrt{\frac{1-P_0}{nP_0}} \tag{2.46}$$

where P_0 is the smallest non-zero probability in the integral that we will need to estimate (we never have to worry about actual 0 probabilities because they drop out of the information integral) then:

$$\hat{I}(s^C, a^C; r) = \int \left[(P_1) \log \left[\frac{P_2}{P_3} \right] \right] + O\Big(\sqrt{\frac{1}{4n}} \log \left[1 + \sqrt{\frac{1-P_0}{nP_0}} \right] \Big) \tag{2.47}$$

Obviously, this integral cannot be performed in a continuous fashion. Instead, we will take the standard numerical integration technique of dividing the integration space

into multi-dimensional boxes (each h units on a side) and computing the integral over those boxes. That is, let us divide the k-dimensional integration space into a set of $(1/h)^k$ multi-dimensional bins $\mathcal{B} = \{\mathcal{B}_1, \ldots \mathcal{B}_{(1/h)^k}\}$ such that the integral decomposes into:

$$\hat{I}(s^C, a^C; r) = I(s^C, a^C; r) + \epsilon_{int} = A \tag{2.48}$$

$$A = \sum_{\mathcal{B}_i \in \mathcal{B}} h^k \left[P_1(\mathcal{B}_i) \log \left[\frac{P_2(\mathcal{B}_i)}{P_3(\mathcal{B}_i)} \right] \right] + O\Big(\sqrt{\frac{1}{4n}} \log \left[1 + \sqrt{\frac{1 - P_0}{nP_0}} \right] \Big) \tag{2.49}$$

where $P(\mathcal{B}_i)$ is the probability that one of the training tuples falls in the mutli-dimensional bin $\mathcal{B}_i$. Effectively we are sampling the multi-dimensional probability distribution at the center of each box, and the error of this expression can be determined by the standard Newton-Cotes formula for numerical integration errors. Here we will use the zero-eth order Newton-Cotes forumla (approximating each bin by a constant value) which yields:

$$\epsilon_{int} = \frac{h^3}{24} |f''(\xi)| \tag{2.50}$$

where:

$$f = P_1 \log \frac{P_2}{P_3} \tag{2.51}$$

and ξ is as usual the value which maximizes the error. Finally, we have

$$\epsilon_{total} < \left[\frac{h^3}{24} |f''(\xi)| + O\Big(\sqrt{\frac{1}{4n}} \log \left[1 + \sqrt{\frac{1 - P_0}{nP_0}} \right] \Big) \right] \tag{2.52}$$

Now, although the exact value of P_0 will depend on the nature of the agent system we deal with, we should have that:

$$P_0 \propto h^k \tag{2.53}$$

as we take smaller and smaller bins, the minimum non-zero probability in any bin will shrink. Thus, we can write:

$$\epsilon_{total} < \left[\frac{h^3}{24}|f''(\xi)| + O\Big(\sqrt{\frac{1}{4n}}\log\left[1 + \sqrt{\frac{1-h^k}{nh^k}}\right]\Big)\right] \tag{2.54}$$

■

The most interesting aspect about the error term is that while decreasing bin sizes helps the *integration* error, it hurts the probability sampling error. Fortunately however the sampling error grows logarithmically such that shrinking the bin sizes by a given amount only has a logarithmic effect on the total error.

H. Error and Policy Quality

We now want to look and see how the quality of our policy correlates with the amount of information contained in a coalition. Specifically, after the last section, we showed how error could enter into the information integral - a natural question is "How much would this error affect final policy quality?".

As a first step in this direction, we will introduce a theorem relating the information content of a coalition to policy quality, *for a specific class of systems*. Specifically, we will introduce a new definition, *invariantly Markov*, which will describe an ide-

alized environment, and will then proceed to prove a theorem which applies to such systems.

1. Coalitions and the Markov Property

As a first step, imagine that we have a system such that the environment of each coalition to be Markovian no matter which agents are in that coalition. This is not unreasonable, and in fact is assumed by training algorithms such as LSPI, and was included in our earlier assumptions in the initial chapters of this dissertation. However, this requirement has some subtle consequences which warrant mention.

Definition 1: Reducibly Markov. An agent system is reducibly Markov if for any agent coalition C, the state transition model $P(s, a, s')$ of that coalition C is an MDP such that $\partial P(s, a, s')/\partial t = 0$ for all times t.

2. Deviations

For most system this is technically rarely the case. That is, the transition model for any given agent will depend on the actions of all other system agents. Consider the simple case of robot exploration, where a system of robots navigates an environment in order to map it out. Even here the transition model of each agent is affect by the actions of the other agents - for example, there may be collisions, or other robots which block narrow hallways, etc.

3. A More Stringent Requirement

A more demanding extension of *reducibly Markov* is that the transition model of an agent coalition not be affected by the behavior of other system agents. Note that this does not imply that *reward* is not dependent jointly on the actions of all system agents, only that the way each agent can move around in its environment is not affected by the policies of other agents.

Definition 2: Invariantly Markov. An agent system is invariantly Markov if for any agent coalition C, the environmental transition function of that coalition $P(s^C, a^C, s'^C)$ can be modeled exactly by a Markov decision process M such that $\partial M/\partial \pi_{n_i} = 0$ for all $n_i \notin C$.

Theorem 4: Let our agent system be invariantly Markov. Let C_0 be a coalition that has its KL measure $D(C_0)$ maximized with respect to single-agent additions and has been trained to some policy π^{C_0}. Let C be some coalition of agents derived from C_0 by a series of agent removals. If our coalition system is invariantly Markov then:

$$\Delta\rho = \rho(\pi^{C_0}) - \rho(\pi^C) \leq \frac{(I_0 - H_0) - (I - 1)}{(I_0 - H_0)} \tag{2.55}$$

Where H_0 is the entropy of $P_0 = P(s^{C_0}, a^{C_0}, r)$, I is the mutual information of C, and I_0 is the mutual information of C_0.

Proof: Because our system is invariantly Markov, as we subtract agents from C_0 our environmental transition model will not change, and the performance of the agents remaining in C_0 is independent of removed agents. Thus, we assign the removed agents to the pure random policy. This may result in an overall performance decrease. In

the worst case, this could be:

$$P(s^C, a^C, r) \propto \delta(r) = P_1 \tag{2.56}$$

where δ is the Dirac delta function (in other words, no non-zero reward is ever given out for anything). This is the worst-case state/action/reward function for *any* coalition (if the system has negative rewards we may simply translate all reward until the lowest possible reward is 0).

Now examine what happens when we remove an agent n_i from coalition C_0 to form C. With a probability $1/|A|$ the agent n_i will execute the action from the joint policy π^{C_0} anyway, but with probability $1 - 1/|A|$ the agent will execute a different (random) action. In these cases we will say that the system can do no worse than if it adopted the worst-case state/action/reward distribution.

$$P(s^C, a^C, r) = \frac{1}{|A|} P_0(s^C, a^C, r) + \left[1 - \frac{1}{|A|}\right] P_1 \tag{2.57}$$

This extends simply for the case of multiple agent removals

$$P(s^C, a^C, r) = \frac{1}{|A|}^n P_0(s^C, a^C, r) + \left[1 - \frac{1}{|A|}^n\right] P_1 \tag{2.58}$$

Instead of carrying around the n-dependent notation, we will simply rewrite this as

$$P(s^C, a^C, r) = (1 - g)P_0(s^C, a^C, r) + gP_1(s^C, a^C, r) \tag{2.59}$$

where

$$0 \leq g \leq 1 \tag{2.60}$$

so that

$$R(s,a) = (1-g)\int_r rP_0 + g\int_r rP_1 = (1-g)R_0 + gR_1 \tag{2.61}$$

Now let us look at the difference in mutual information between the optimal coalition C_0 and the coalition C:

$$\Delta I = I(C_0) - I(C) = I_0 - I \tag{2.62}$$

$$\Delta I = \int P_0 log\frac{P_0}{P_B} - \int [(1-g)P_0 + gP_1]\, log\left[\frac{P_0(1-g) + gP_1}{P_B}\right] \tag{2.63}$$

where $P_B = P(r)\prod_i P(s^{n_i})P(a^{n_i})$. Using the equality $log(x+y) = log(x) + log(1+y/x)$ we can reduce this expression to:

$$\Delta I = \int P_0 \log P_0 + \int A + \int B + \int C + \int D + \int E \tag{2.64}$$

where the integrals are taken over states, actions and reward, and:

$$A = -(1-g)P_0 \log P_0(1-c) \tag{2.65}$$

$$B = -(1-g)P_0 \log\left[\frac{P_1 g + (1-g)P_0}{P_0(1-g)}\right] \tag{2.66}$$

$$C = -gP_1 \log P_1 g \tag{2.67}$$

$$D = -gP_1 \log\left[\frac{P_1 g + (1-g)P_0}{P_1 g}\right] \tag{2.68}$$

$$E = -P_0 \log P_B + [(1-g)P_0 + gP_1] \log P_B \tag{2.69}$$

We can make the expression an inequality by removing some always-positive expressions from the right-hand side

$$\Delta I \geq \int P_0 \log P_0 + \int A' + \int B' + \int C' + \int D' + \int E \tag{2.70}$$

where

$$A' = -(1-g)P_0 \log P_0 \tag{2.71}$$

$$B' = (1-g)P_0 \log [P_0(1-g)] \tag{2.72}$$

$$C' = gP_1 \log P_1 g \tag{2.73}$$

$$D' = gP_1 \log [P_1 g] \tag{2.74}$$

$$\Delta I \geq \big(1 - (1-g)\big) \int P_0 \log \frac{P_0}{P_B} - \int gP_1 \log \frac{P_1}{P_B} + K \tag{2.75}$$

$$K = (1-c)H_0 + cH_1 + (1-g)\log(1-g) + g \log g \tag{2.76}$$

where H_0, H_1 are the entropies of the two distributions

$$H_0 = \int P_0 \log P_0 \tag{2.77}$$

$$H_1 = \int P_1 \log P_1 \tag{2.78}$$

or

$$\Delta I \geq gI_0 - gI_1 + (1-g)H_0 + gH_1 + (1-g)\log(1-g) + g\log g \tag{2.79}$$

since $-1 \leq (1-g)\log(1-g) + g\log g \leq 0$ we have:

$$\Delta I \geq gI_0 - gI_1 + (1-g)H_0 + gH_1 - 1 \tag{2.80}$$

$$I_0 - I \geq gI_0 - gI_1 + (1-g)H_0 + gH_1 - 1 \tag{2.81}$$

$$I_0 - I - (H_0 - 1) \geq g(I_0 - I_1 - H_0 + H_1) \tag{2.82}$$

$$\frac{I_0 - I - (H_0 - 1)}{I_0 - I_1 - H_0 + H_1} \geq g \tag{2.83}$$

however, since we chose a Dirac-delta function for our worst-case distribution P_1, we have $I_1 = 0$, and $H_1 = 0$, such that:

$$\frac{(I_0 - H_0) - (I - 1)}{(I_0 - H_0)} \geq g \tag{2.84}$$

and finally, we have:

$$\frac{R_0 - R}{R_0} \leq \frac{(I_0 - H_0) - (I - 1)}{(I_0 - H_0)} \tag{2.85}$$

Since policy quality is directly related to reward

$$\Delta\rho = \rho(\pi^{C_0}) - \rho(\pi^{C}) \tag{2.86}$$

$$\Delta\rho = \sum_{t=0}^{\infty}(R_t^{C_0} - R_t^{C})\gamma^t \tag{2.87}$$

$$\Delta\rho \leq \sum_{t=0}^{\infty}(R_t^{C_0} - R_t^{C_0}\left[1 - \frac{(I_0 - H_0) - (I - 1)}{(I_0 - H_0)}\right])\gamma^t \tag{2.88}$$

$$\Delta\rho \leq \sum_{t=0}^{\infty}(R_t^{C_0}\frac{(I_0 - H_0) - (I - 1)}{(I_0 - H_0)})\gamma^t \tag{2.89}$$

$$\Delta\rho \leq \frac{(I_0 - H_0) - (I - 1)}{(I_0 - H_0)} \tag{2.90}$$

■

4. Discussion

Theorem 4 provides a bound on how much deviations from optimal information affect policy quality. Since **Theorem 2** says that the *only* way we can negatively affect policy quality is by agent removals, and the result was entirely in terms of the information content of the altered coalition C, **Theorem 4** gives us a bound on how much worse the coalition may perform if its mutual information is not actually maximized.

This maximization may fail for a variety of reasons. It may be the case that errors in the information integral mislead us into *not* including an appropriate agent in a coalition, or maximum coalition size restraints may prevent us from maximizing

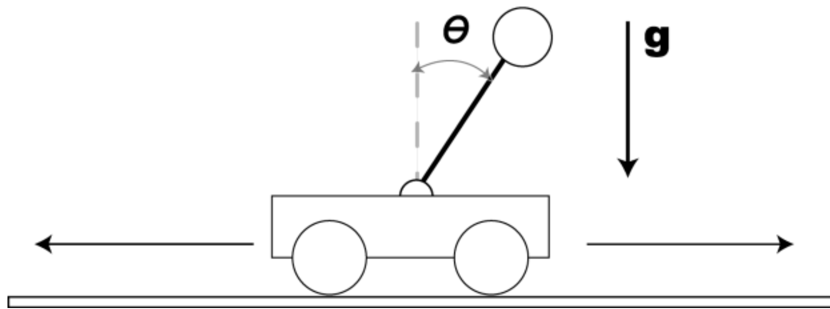

Fig. 1. The cart-pole balancing problem.

mutual information. Either way, this theorem gives us a bound on how much these errors will affect final policy quality. Specifically, it is reassuring that the error only scales linearly with information deviations.

It is true that this **Theorem 4** is something of an idealization, since it is proven only for invariantly Markov systems. Future work might be done to expand this theorem to more general domains.

I. Experiment 1: Multi-Agent Cart-Pole Balancing Problem

One of the classic problems in reinforcement learning is the cart-pole balancing problem. In this problem we have a cart of mass M capable of frictionless, one-dimensional motion. Attached to this cart via a hinge is a pole with a mass m on top. Gravity acts to pull the top mass down, but the pole and mass can be kept in the upright position by judicious movements of the cart. Figure 1 illustrates the basic quantities associated with this problem.

Agents in this system are capable of applying some impulse either to the left or to the right of the cart, and reward is given out to the agents if the pole is within some angle tolerance of the upright position. System parameters are specified in two

variables, θ, the angle of the pole away from the upright position, and $\omega = \dot{\theta}$, the angular velocity of the pole.

We performed a multi-agent version of this simulation to demonstrate our coalition formation algorithm. Specifically, in our system, each agent was given a single sensor which measured some linear combination of θ and ω (to enable direct combination of these two variables, both θ and ω were normalized to have zero mean and unit variance before use in all cases). Specifically, each agent was given a single sensor η_i:

$$\eta_i = cos(\phi_i)\theta + sin(\phi_i)\omega \tag{2.91}$$

And as usual, each agent was capable of imparting a force from the left or right to the cart. This problem is interesting, because individually, each agent has insufficient information to solve the pole-balancing problem. That is, neither θ or ω alone is enough to solve the pole balancing problem, and neither is any linear combination of these two variables. Agents must be in coalitions such that other agents supply missing pieces of information to allow both agents to come to a joint decision as to what force to apply to the cart.

Pre-training, we ran our IBCF algorithm as described above on the cart-pole agents. Intuitively, one would expect that the maximum amount of reward-related information would occur in two-agent coalitions $\{n_1, n_2\}$ (as in Figure 2) such that $\{\phi_1 = x, \phi_2 = x + \pi/2\}$, with higher-agent coalitions offering no informational advantage. That is, such a two-agent coalition has all the information contained in the original θ and ω variables - and in fact, the two variables θ and ω can be recovered by a rotation of $\{\eta_1, \eta_2\}$ by an angle $-x$.

We collected a training corpus by allowing $N = 8$ agents to interact with a pole-cart system. All physical constants and simulation methods were taken directly

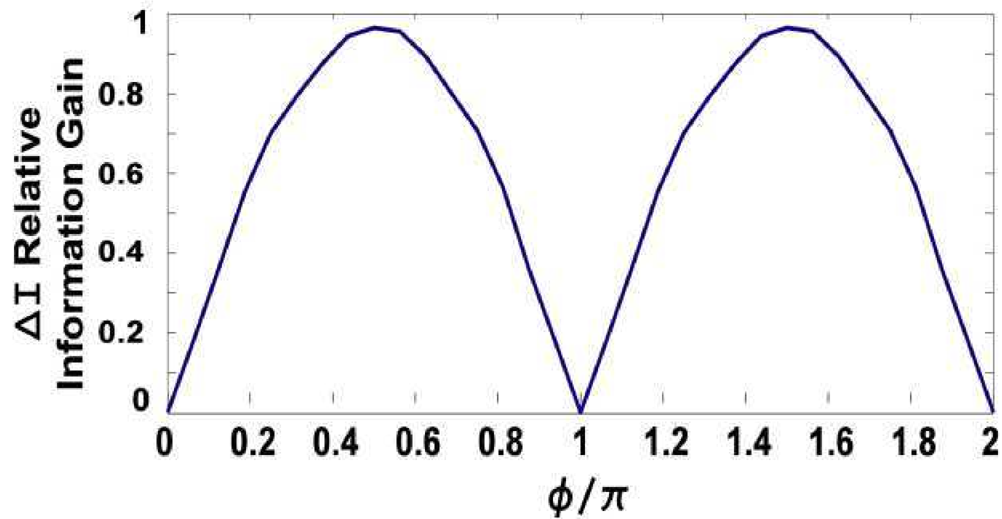

Fig. 2. Relative information example. Relative information gain between the single-agent coalitions $C_1 = \{n_1\}, C_2 = \{n_2\}$ with $\phi_1 = 0, \phi_2 = \phi_1 + \phi$, and the two-agent coalition $C_3 = \{n_1, n_2\}$. Specifically, $\Delta I = \frac{I(C_3)-(I(C_1)+I(C_2))/2}{(I(C_1)+I(C_2))/2}$. Notice that when $\phi = 0$ the sensors of the two agents are identical and no information gain is achieved. The maximum is achieved at $\phi = \pi/2$ when the two sensor outputs are 'orthogonal'.

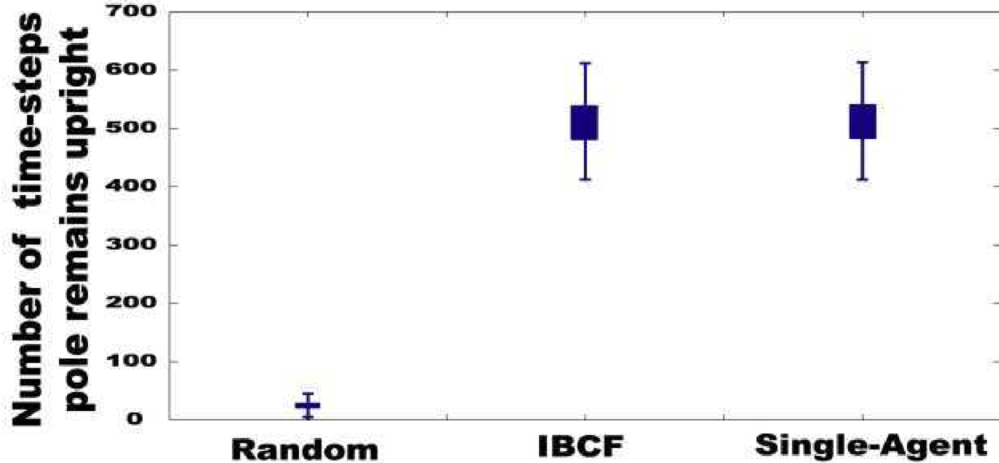

Fig. 3. Algorithm comparison with fixed network connectivity.

Table II. Results of the cart-pole balancing experiment, $\pm$ one STD shown

Algorithm	Upright timesteps
Random policy	30 ± 24
CRL	507 ± 102
IBCF	505 ± 105

from the single-agent pole-balancing example presented in [23]. During experience collection, agents followed a random policy, applying force from the left or right with equal probability. We initialized the sensor angles to be the following $\phi_1 = 0, \phi_2 = \pi/2, \phi_3 = \pi/8, \phi_4 = \pi/8+\pi/2, \phi_5 = \pi/4, \phi_6 = \pi/4+\pi/2, \phi_7 = 3\pi/4, \phi_8 = 3\pi/4+\pi/2$. After the training corpus was collected (we collected 1600 tuples, or 200 per agent, as in the original Lagoudakis and Parr paper), we ran IBCF over the corpus, which produced the following coalition structure:

$$\mathcal{CS}_{\text{IBCF}} = \{\{n_1, n_2\}, \{n_3, n_4\}, \{n_5, n_6\}, \{n_7, n_8\}\} \tag{2.92}$$

As expected, the algorithm matched agents with complementary sensors such that intra-coalition mutual information between states/actions and reward was maximized. After coalition structure generation, we trained each coalition in isolation to balance the pole using the reinforcement learning scheme described in [23] (Least-Squares Policy Iteration). After training was complete, we measured the average time the system was able to keep the pole balanced upright (with all agents in all coalitions able to apply forces simultaneously, of course). For comparison, we also measured the average 'upright time' when randomly generated coalition structures (made up of strictly two-agent coalitions) were used, and the average upright time for a single agent balancing the pole with access with both θ and ω directly (essentially reproducing the single-angle case presented in [23] ... in this domain, the single-agent policy will perform optimally). For each case, we allowed the system to try to balance the pole 500 times, and the average upright times are plotted in Figure 3 and listed in Table II. The IBCF-coalition based results are nearly identical to the single-agent results, as expected, since each coalition has as much information about the system as does the single agent. Note that we did not compare against a direct search for an optimal coalition structure because the rather long training/evaluation times for this domain made this untenable.

J. Experiment 2: Multi-Agent Network Control Problem (SysAdmin)

The SysAdmin problem, as presented in [59] simulates the management of network by a system of agents. The problem roughly goes as follows: there is a network of N machines (connected to each other through some network topology), each of which is managed by an agent. These machines are designed to run processes, and when a

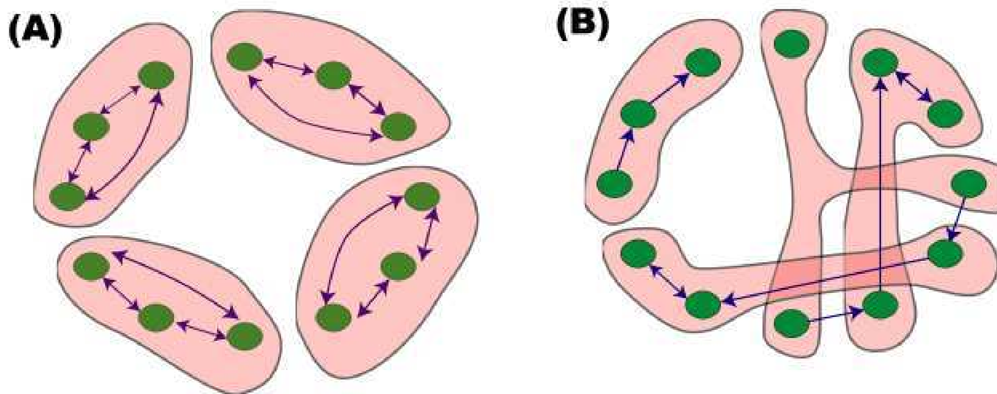

Fig. 4. Example network topologies. Two network topologies with the coalitions produced by IBCF drawn on the figure.

machine completes a process, its agent is given some reward. Specifically, machines are capable of being in three 'stability' states STATE = {HEALTHY, UNSTABLE, DEAD} and three 'process' states LOAD = {IDLE, PROCSES RUNNING, PROCSS COMPLETE}. Healthy machines turn into unstable machines with some probability, and unstable machines turn into dead machines with some probability. Unstable machines take longer to complete processes (and hence generate less reward on average) and dead machines cannot run processes at all. To make matters more complicated, unstable and dead machines can send bad packets to their neighbors, causing them to go unstable and eventually die as well. At each time-step, each agent must decide whether or not to reboot its machine. Rebooting returns the machine to the healthy state, but at the cost of losing all running process (which effectively incurs some reward penalty since all work done on processes up to this point will be lost).

For our experiment, we generated machine networks with various numbers of nodes and with randomly generated network topologies. Specifically, to generate a network of size N with a connectivity of m we first set up our N nodes and then randomly connected each node to m other nodes in the network.

For each network, we collected a training corpus by having each agent follow a

random policy and reboot its machine with a 50% probability. All other simulation parameters were as described in the [59] work. To speed up training, we capped our coalition sizes at three agents in all cases. After coalition generation, we trained each coalition in isolation, using Least-Squares Policy Iteration, as outlined in [23].

It is interesting to observe the coalitions that our algorithm generated for various network topologies. Notice in Figure 4a that for a system where nodes are strictly connected to two neighbors (and an appropriate three-agent coalition structure is obvious), our algorithm generates coalitions reflecting these dependencies. Other more complex topologies resulted in different coalition structures, for example in Figure 4b, though the interdependencies of the network are still somewhat represented.

First, we examined how the algorithm performed as a function of network connectivity. Since in all cases we capped the maximum coalition size at 3, as the network nodes became increasingly interconnected, one would expect post-training performance of the system to drop off. One would expect that knowledge of the state of nodes connected to an agent's node would assist that agent in choosing a more optimal restart policy.

Look at Figure 5, which shows post-training average reward as a function of the number of connections each node had (these results are also listed in Table III). Notice that although for sparse networks (less than 2 connections per node) the algorithm performs relatively well. Although the general trend is for reward to drop off with network connectivity (as should be expected since more connections means more opportunities for faulty nodes to spread their faulty state to their neighbors), there is a large drop-off from 2 connections per node to 3 connections per node. This makes sense in the context of our algorithm - with each node connected to only two other nodes, each node need only concern itself with its own state plus the state of its two neighbors, and thus a 3-agent coalition was generally sufficient to group

agents together. In the situation where we have each node connected to 3 other nodes however, each node must concern itself with its own state, plus the state of its three other neighbor nodes, which suggests that a 4-agent coalition would have better served the situation.

To compare our algorithm's performance against CRL, we also generated a series of 600 random-topology, 12-node networks such that each node was connected to exactly two other nodes. This replicated the test networks used to benchmark CRL in [59]. We ran our IBCF algorithm on these networks, first generating coalitions, and then using LSPI to train each coalition separately. For further comparison, we also formed random coalitions and then used LSPI to train each random coalition separately. In Figure 6 (Table IV) we see that our algorithm produced coalitions that significantly outperformed the random coalitions, and the IBCF-based system also came close to the performance of the CRL algorithm.

Since the overall reason for training agents in coalitions in the first place was to reduce training times, we explored the training times required to train networks of various sizes. In this case as well we capped the coalition size to 3 nodes. We generated random networks of various sizes (for each size we generated 500 random topologies), and in all cases each network node was connected to exactly two other nodes (not that this would affect the time required to train a policy). For comparison we monitored the time to train each network with CRL and with our combination of IBCF and then LSPI over each generated coalition. Additionally, we trained each network with standard LSPI over all network nodes simultaneously. Look at Figure 7 and Table V. We see that our IBCF-based approach and the CRL algorithm both have training times that do not scale exponentially with agent count, though, as expected, the training times for naive system-wide LSPI scaled exponentially with agent count. Notice also that our algorithm was slightly slower than CRL in all cases. This is because whereas CRL assumes that the system designer has effectively already divided the agents into coalitions and can thus immediately begin to train the system. In contrast, we must first perform a pass of IBCF to form our coalitions and then proceed to perform an LSPI pass to train each coalition.

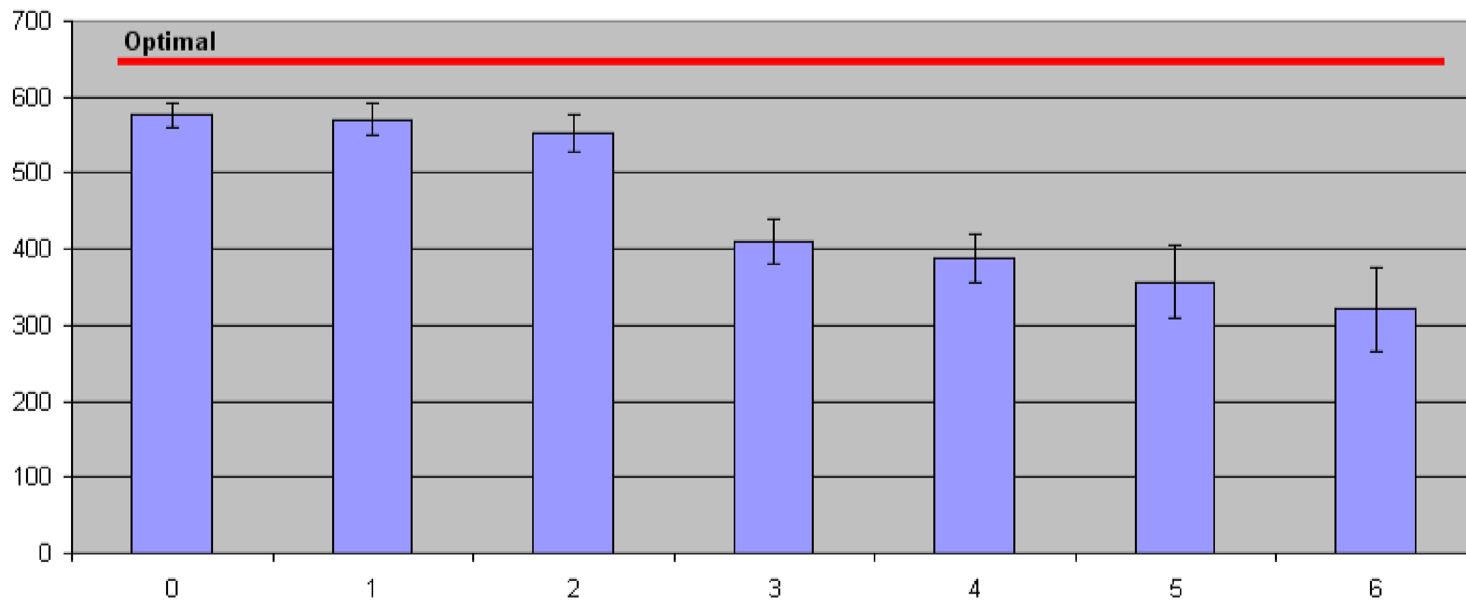

Fig. 5. Algorithm performance as a function of network connectivity (number of connections per node).

Table III. Algorithm performance on the network control problem as a function of connected nodes

Neighbor Nodes	Average Reward	Standard Deviation
0	576	15
1	571	21
2	552	24
3	410	29
4	388	31
5	357	48
6	321	56

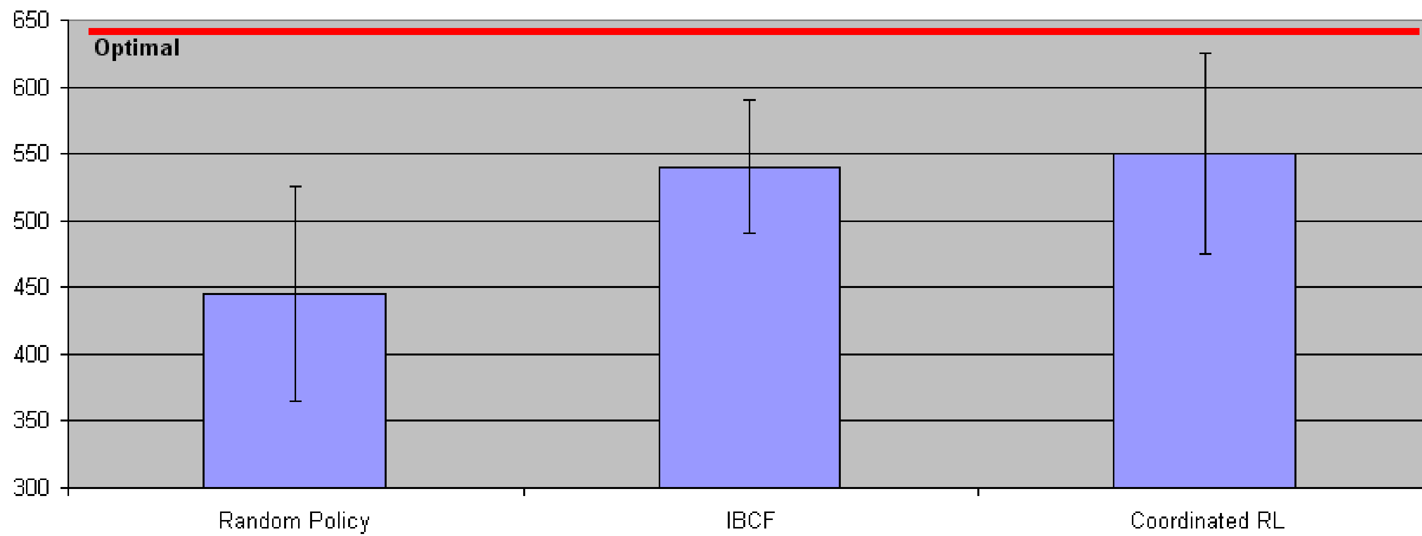

Fig. 6. Results of the SysAdmin simulation. Performance of the various algorithms on the 2-neighbor, 12-node network control problem.

Table IV. Performance comparisons on the 12-node, 2 connection set

Algorithm	Long-Term Reward Mean	Standard Deviation
Random Policy	450	100
IBCF	554	24
Coordinated RL	570	50

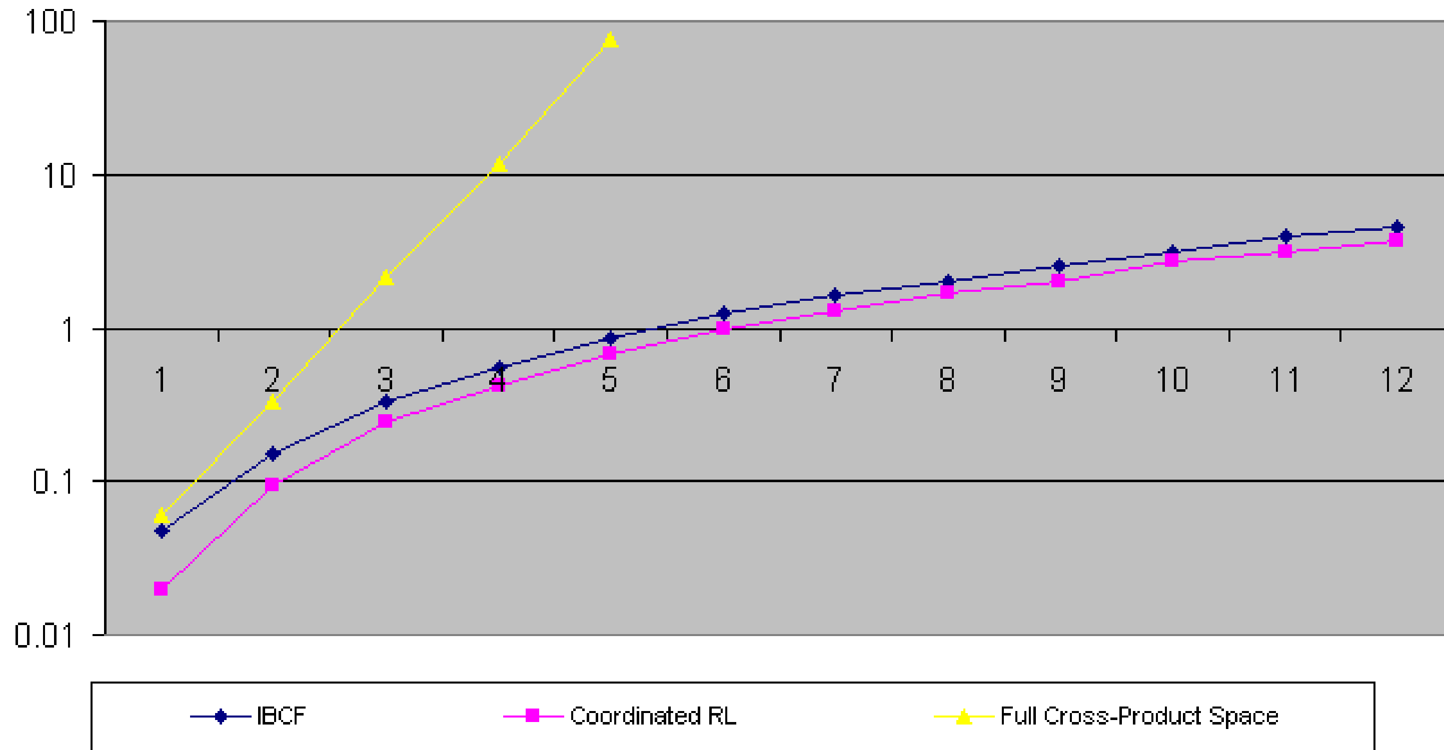

Fig. 7. Average training times (in minutes) for the various algorithms as a function of network size.

1. Discussion

Our results show that our IBCF algorithm performed admirably against coordinated RL, even though our algorithm had access to far less information, and additionally, generated coalition structures before training took place.

Additionally, as expected, as the complexity of our networks increased, our algorithm performed less well. Specifically, when we increased the number of nearest neighbors past our maximum coalition size, network performance decreased sharply. Notice that the network's performance was even worse than when the random policy was applied. This is a little misleading. In the first experiment, where we compared the performance of IBCF against CRL, our networks were relatively simple. By increasing the connectivity of our networks, we also increased the rate at which errors could be transmitted, inherently lowering network performance. Another side-effect of this is that lower-complexity networks (see the 1-nearest-neighbor results) inherently performed better, in part because there were fewer linkages to transmit faults.

The important result here is that there was a sharp drop-off in network performance after the demands of the domain exceeded the coalitional capacity. We interpret this as the case where we are unable to maximize mutual information (additional agents would have increased the mutual information) and generate sub-optimal coalitions. This is in some extent the point, since the maximum coalition size was intentionally capped below the nearest neighbor count. Note that the algorithm's performance did decrease gracefully beyond the 2-neighbor point. Lower neighbor networks generally performed better because there were fewer linkages and thus a lower probability of being 'infected' by connected faulty nodes.

Table V. Average training times (minutes)

Network Size	Full Network LSPI	Coordinated RL	IBCF + LSPI
1	0.061	0.019	0.048
2	0.332	0.094	0.150
3	2.157	0.240	0.332
4	11.824	0.422	0.553
5	76.283	0.667	0.847
6	-	0.985	1.259
7	-	1.312	1.659
8	-	1.679	2.006
9	-	1.998	2.544
10	-	2.701	3.149
11	-	3.150	3.935
12	-	3.733	4.582

Table VI. Results of the power-grid experiment

Network	Random	CRL	ICBF
A	29.7 ± 0.13	0.08 ± 0.01	0.07 ± 0.02
B	52.00 ± 0.24	0.13 ± 0.02	0.15 ± 0.20
C	96.8 ± 0.31	40.86 ± 1.14	45.21 ± 2.12
D	44.14 ± 0.37	0.11 ± 0.02	0.12 ± 0.03

K. Experiment 3: Power Distribution

We also tested our algorithm on a third domain, Schneider's *power grid* domain [67]. In this domain one has a network (much like the SysAdmin domain), only each node can either be a *provider* (a fixed voltage source), a *customer* with a desired voltage, or a *distributor* (where agent control takes place). Links between nodes have resistances, and the distributors must set the resistances to meet the demand of the customers. If a given customer's demand is not met, then the grid is penalized equal to the demand minus supply. The total penalty is simply the sum of the penalties incurred across all customers. At each time-step each distributor can either *double*, *halve*, or *maintain* their resistances (3 possible actions), and there are 6 possible resistance levels.

Structurally, these networks are very similar to the SysAdmin networks from the previous experiment, usually not exceeding each node having 2 neighbors. We ran the same simulator we built for the SysAdmin domain on the power-grid networks (after altering the simulation to accommodate the physical electrical parameters). We compared the results against CRL run on the same network. We used exactly the experimental setup as described in [62]. This called for 10,000 training samples to be generated, and the network to be evaluated for 60,000 time-steps. The results are shown below in Table VI (network topologies are as in [67, 59]:

The experiment showed that our algorithm was again comparable to CRL in terms of post-training performance. The one notable result from this experimental run is network C. Here the complexity of the network exceeds the 2-nearest neighbor motif of the other networks. In [62] the authors use a 2-node basis to represent their environment (that is, they restrict each agent to only observing two of their linked neighbors). This explains why CRL does worse on this network, and similarly, since we limited ourselves to coalitions of 3 agents or less, our algorithm fared poorly on this higher-complexity network.

L. Conclusions

We have present a unique algorithm capable of generating high-performance agent coalitions *before* any training occurs, and *without* knowing any a priori information about agent relationships.

Our approach was to form coalitions to maximize mutual information, and we provided a basic theoretical justification for this approach. We also derived error bounds which relate deviations from maximal coalition information to policy quality decreases. We found the relationship between information and policy quality to be linear.

This approach is inherently an approximation in the sense that post-coalition-formation we train over groups of separated agents and are not guaranteed to achieve the globally optimal policy.

We applied our algorithm to three multi-agent reinforcement domains: the multi-agent pole-balancing problem (as in [23]) and the SysAdmin, network management problem (as in [59]), and a power-grid management domain (see [**?** 59]. In all cases

our algorithm performed comparably to coordinated reinforcement learning. We noticed performance drop offs whenever the system complexity exceeded our maximum coalition size. This was a trend in both network-based domains (SysAdmin and power grid), and should be expected. In such cases we hypothesize that the algorithm is not able to maximize mutual information - additional agents would significantly increase the coalition's knowledge of how states and actions relate to reward. Raising the coalition size limit would probably solve this problem, though note that even when system complexity grew too high, our algorithm failed gracefully, as would be expected from a non-exponential (linear in fact) relationship between information and policy quality.

CHAPTER III

INTEGRATING FUZZY KNOWLEDGE INTO REINFORCEMENT LEARNING

Reinforcement learning is hard. Given a set of (possibly noisy) examples of how an agent's actions affect the environment, one must essentially produce a *model* of the potentially very complex interactions between agent actions, environmental response, and external rewards distributed to the agent from a series of training examples. As we discuss in this chapter, the *sample complexity* of reinforcement learning (the number of samples that the learning agent needs to have access to in order to develop a good policy) grows exponentially with the size of the agent system. As such, in many cases it is difficult, if not impossible, to supply the learning system with enough training examples to successfully develop a high-performance policy. Fortunately, this is somewhat of a common problem in machine learning in general. Many learning concepts have sample complexities that are infeasible, and the approach that is generally taken to rectify the situation is to engage in some sort of 'hybrid learning' where, in addition to training examples, the learner is given access to *domain knowledge* [83]. This domain knowledge can augment what is available from the training examples and allow the learner to develop a solution superior to what could be obtained from training examples alone.

Interestingly, there has been a long-standing interest in integrating pre-training domain knowledge into the reinforcement learning process to 'help' the learner out by giving hints about the nature of the learning environment. Historically, this domain knowledge has taken various forms - for example, the work by Mahadevan, et. al. [84] which considered the effect of giving learners knowledge of important environmental sub-tasks to speed up learning, or the work by Milan [85] which examined the effects of giving learning agents reflex rules to focus state-space exploration where it

is most needed. Other examples include Mostafa's work on giving 'hints' to machine learning processes to help out learning [86, 87, 88, 89], and Mataric's work on using "rich reward functions" (modifying a system's reward function to reflect user-provided knowledge) [90, 91], or Hailu's work on how embedding environmental knowledge can ease state space construction [92].

In our research direction however, we have focused on domains where we know little or no information about the training system. As such, we wish to here treat the case where we have some domain knowledge, but this knowledge is at best incomplete, and as such we must express our knowledge in terms of fuzzy logic rules. We will show how to incorporate this knowledge into the RL training paradigm by systematic modification of the system reward function. In the spirit of this dissertation, this constitutes an approximate method since inherently, modification of the reward function introduces some error into training. We quantify this error and demonstrate the use of our technique on several experimental domains. Specifically, we see how the addition of domain knowledge (encoded in simple fuzzy rules) significantly improves the performance of the post-learning pole-balancing controller and significantly reduces the sample complexity on this domain.

Our work will be most closely related to the Mataric work cited above, though we will not explicitly use any of the techniques derived therein. Specifically, whereas previous work on reward-shaping techniques (such as the Mataric work) focused on older reinforcement learning techniques such as Q-learning, here we focus on the use of more modern RL techniques such as Least-Squares Policy Iteration (LSPI) [23]. In that algorithm there are two quantities, $\mathbf{A}$ and $\mathbf{b}$, which are generated solely from experience tuples sampled from the environment. As previously discussed, $\mathbf{A}$ essentially captures the transition model of the environment, while $\mathbf{b}$ captures the dynamics of the environmental reward function. Here we present a method to

incorporate *domain knowledge* (and specifically, fuzzy domain knowledge) into the reward vector **b**.

A. Sample Complexity and Reinforcement Learning

The sample complexity of a concept is the number of training examples that a learner trying to learn the concept needs to be exposed to before the learner can, with high probability, learn the concept. Specifically, in the case of reinforcement learning, the sample complexity has been shown to be the following: In order to guarantee that with probability $1 - \delta$ that one will obtain post-training a policy π such that:

$$||Q^*(s,a) - Q^\pi(s,a)|| \leq \epsilon ||Q^*(s,a)|| \tag{3.1}$$

the learner needs access to:

$$m = O\left(\frac{|A|^n K}{(1-\gamma)\epsilon} ln \frac{|A|^n}{(1-\gamma)\epsilon} + ln\frac{2}{\delta}\right) \tag{3.2}$$

training examples, where $|A|$ is the size of the action space of a single agent, n is the number of system agents, and K is a parameter which depends on the details of the environmental MDP. Notice that like the sample complexity grows exponentially with agent count such that for large systems of multiple agents, it seems likely that the learner will have to learn from inadequate numbers of training examples.

In this chapter we present a framework for overcoming such training example shortages by injecting domain knowledge into the reinforcement learning process. As we said before, given the context of this dissertation we specifically want to concentrate on domain knowledge expressible as fuzzy-logic based rules.

B. Domain Knowledge and Fuzzy Rules

Fuzzy logic is an extension of fuzzy set theory that deals with *approximate* reasoning (in contrast to the absolute reasoning found in classical predicate logic) [93, 94]. Fundamental to fuzzy logic is the notion of *membership functions*. Generally membership functions $h(X)$ are real-valued functions with a range between 0 and 1, and represent the degree to which X belongs to the set that h represents. By convention, higher values of the membership function represent a higher degree of belonging to the associated fuzzy set. For example, we might define the fuzzy set *red* with the associated membership function $h_{red}(blood) = 1$, $h_{red}(grass) = 0$, $h_{red}(sunset) = 0.8$, $h_red(maroon) = 0.5$, etc. The higher the value of h, the more the predicate X belongs to h's associated fuzzy set.

We now wish to explore how domain knowledge (eventually in terms of fuzzy rules) can be integrated into the LSPI learning environment. To begin with, let us imagine that we have expressed what we know about a given learning domain in terms of fuzzy rules (for example "If TOO_CLOSE Then AVOID_COLLISION" in a navigation robot). Specifically, we will take these fuzzy rules F as consisting of a fuzzy predicate over state variables which maps to an agent action:

$$F : \mathcal{S} \mapsto \mathcal{A} \tag{3.3}$$

We can represent F as a fuzzy set by defining a membership function $h_F(s, a)$ (and as usual with fuzzy-logic setups, we will take $0 \leq h_F(s, a) \leq 1$). Here if a (s, a) pair has a high $h_F(s, a)$ it means that taking action a in state s is a good example of the rule F, and conversely if $h_F(s, a)$ is low, it means that performing a in state s is a bad example of the fuzzy rule F.

At some point, we would like to encourage a learner to take advantage of this

rule. Taking inspiration from the work on rich-reward functions by Mataric [90], we will primarily seek to accomplish this by modifying the reward function $R(s,a)$ (where C is some constant):

$$R'(s,a) = R(s,a) + C\, h_F(s,a) \tag{3.4}$$

That is, we perform rule integration by offering the learner some reward proportional to how well a given state action pair is an example of the rule F.

C. Potential Negative Effects of Added Domain Knowledge

1. Derivation

In this chapter, we intend to use reward shaping to integrate domain knowledge into the reinforcement learning process. However, we must at some point recognize a somewhat obvious potential danger: after using reward shaping on the reward function R, when we train an agent using reinforcement learning, we train on a *modified* reward function R'. This reward function R' is not the one which was established by the environment (after all, we just shaped it with helpful fuzzy domain knowledge), and this should give us pause.

In fact, we saw that depending on *how* the reward function is modified, the post-training system can exhibit vastly different behaviors with vastly different quality measures. Even if we shape the reward function in a way that we *think* will result in better performance, there is still the question of how *much* shaping we can or should do.

A more abstract version (and one that will be amenable to analysis) is to ask

what is the *worst cast* performance hit that we could take by modifying our reward function by some given amount. Fortunately, this question can be answered in a fairly straightforward manner due to some theorems that were proved by Bertsekas [95]:

Theorem 1 (Bertsekas): Let $\hat{\pi}_0, \hat{\pi}_1, \hat{\pi}_2, \ldots, \hat{\pi}_m$ be the sequence of policies generated by an approximate policy iteration algorithm and let $\hat{Q}^{\hat{\pi}_0}, \hat{Q}^{\hat{\pi}_1}, \hat{Q}^{\hat{\pi}_2}, \ldots, \hat{Q}^{\hat{\pi}_m}$ be the corresponding approximate value functions. Let ϵ and δ be positive scalars that bound the error in all approximations (over all iterations) to value functions and policies respectively. If

$$\forall m = 0, 1, 2, \ldots, \ ||\hat{Q}^{\hat{\pi}_m} - Q^{\hat{\pi}_m}||_\infty \leq \epsilon \tag{3.5}$$

and

$$\forall m = 0, 1, 2, \ldots, \ ||T_{\hat{\pi}_{m+1}} \hat{Q}^{\hat{\pi}_m} - T_* \hat{Q}^{\hat{\pi}_m}||_\infty \leq \delta \tag{3.6}$$

where T is the Bellman optimality operator defined as:

$$(T_*)Q(s,a) = R(s,a) + \gamma \sum_{s' \in S} \mathcal{P}(s,a,s') \max_{a' \in \mathcal{A}} Q(s',a') \tag{3.7}$$

Then, this sequence eventually produces policies whose performance is at most a constant multiple of ϵ and δ away from the optimal performance:

$$\lim_{m \to \infty} ||\hat{Q}^{\hat{\pi}_m} - Q^*||_\infty \leq \frac{\delta + 2\gamma\epsilon}{(1-\gamma)^2} \tag{3.8}$$

■

This theorem makes guarantees about the final quality of a policy trained under policy iteration when the Q function undergoes a small perturbation (potentially a different one at each iteration of training). The theorem above was originally intended to be used in an approximate policy iteration setting, where the value function Q is represented by linear combinations of basis functions, and the 'perturbations' are caused by approximation error. However, the theorem generally applies to any situation where there is some finite 'error' introduced to the Q function.

Theorem 2: Let R be a reward function. If we modify R as $R' = R + \Delta R$ such that $\forall s, a \ \ |\Delta R| < \epsilon_R$ and run LSPI on the modified reward function, we will have:

$$\lim_{m \to \infty} ||\hat{Q}^{\hat{\pi}_m} - Q^*||_\infty \leq \frac{2\gamma(\epsilon_A + \epsilon_R/(1-\gamma))}{(1-\gamma)^2} \tag{3.9}$$

where Q^* is the discounted reward of the optimal policy for the reward function R, and ϵ_A is the approximation error from the Bertsekas proof (Theorem 1).

Proof: If we modify R' as above, and let Q' be the discounted reward under the reward function R', we have (by the definition of discounted reward):

$$Q'^{\pi}(s,a) = R'(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s,a,s') Q'^{\pi}(s', \pi(s')) \tag{3.10}$$

This expression can be re-expressed in the following form (by repeatedly using the expression for discounted reward):

$$Q'^{\pi}(s,a) = R'(s,a) + \sum_{i=1}^{\infty} \gamma^i \left(\sum_{s' \in \mathcal{S}} R'(s', \pi(s')) P_i^{\pi}(s,a,s') \right) \tag{3.11}$$

Where $P_i^{\pi}(s,a,s')$ is the probability of being in state s' after starting in state s

and taking action a, then following the policy π for i actions. Since $R'(s,a) = R(s,a) + \Delta R(s,a)$, we have:

$$Q'^{\pi}(s,a) = R(s,a) + \Delta R(s,a) + \sum_{i=1}^{\infty} \gamma^i \left(\sum_{s' \in \mathcal{S}} (R(s', \pi(s')) + \Delta R(s', \pi(s'))) P_i^{\pi}(s,a,s') \right) \tag{3.12}$$

Letting

$$\Delta Q^{\pi}(s,a) = \Delta R(s,a) + \sum_{i=1}^{\infty} \gamma^i \left(\sum_{s' \in \mathcal{S}} \Delta R(s', \pi(s')) P_i^{\pi}(s,a,s') \right) \tag{3.13}$$

we also have

$$Q'^{\pi}(s,a) = Q^{\pi}(s,a) + \Delta Q^{\pi}(s,a) \tag{3.14}$$

However, by assumption $\forall s,a \ \ ||\Delta R(s,a)|| < \epsilon_R$, such that:

$$\forall s,a \ \ ||\Delta Q^{\pi}(s,a)|| < \frac{\epsilon_R}{1-\gamma} \tag{3.15}$$

Such that:

$$\forall s,a \ \ |Q'^{\pi}(s,a) - Q^{\pi}(s,a)| < \frac{\epsilon_R}{1-\gamma} \tag{3.16}$$

for any policy π ... in particular, this applies to the policies produced by approximate policy iteration:

$$\forall s,a \ \ \forall m = 0,1,2,\ldots, \ \ |Q'^{\hat{\pi}_m}(s,a) - Q^{\hat{\pi}_m}(s,a)| < \frac{\epsilon_R}{1-\gamma} \tag{3.17}$$

Where, just to be clear, $Q'^{\hat{\pi}_m}(s,a)$ is the discounted reward obtained from following policy $\hat{\pi}_m$ under the reward function $R' = R + \Delta R$ and $Q^{\hat{\pi}_m}(s,a)$ is the discounted reward obtained from following the same policy under the unmodified reward function R. Thus, if we have, for some fixed ϵ_A (assumed in **Theorem 1**):

$$\forall m = 0, 1, 2, \ldots, \; ||\hat{Q}'^{\hat{\pi}_m} - Q'^{\hat{\pi}_m}||_\infty \le \epsilon_A \tag{3.18}$$

then we have:

$$\forall m = 0, 1, 2, \ldots, \; ||\hat{Q}'^{\hat{\pi}_m} - Q^{\hat{\pi}_m}||_\infty \le \epsilon_A + \frac{\epsilon_R}{1-\gamma} \tag{3.19}$$

such that **Theorem 1** is satisfied with $\epsilon = \epsilon_A + \frac{\epsilon_R}{1-\gamma}$ so that in the limit of policy iteration we have:

$$\lim_{m \to \infty} ||\hat{Q}^{\hat{\pi}_m} - Q^*||_\infty \le \frac{2\gamma(\epsilon_A + \frac{\epsilon_R}{1-\gamma})}{(1-\gamma)^2} \tag{3.20}$$

■

2. Interpretation

Theorem 2 has several properties worthy of discussion. First, we see that optimality bound is proportional to the amount of reward shaping ϵ_R. This makes intuitive sense that the more we alter the reward function, the more we risk training a sub-optimal policy. Second, we see that the bound is *highly* dependent on the discounting factor γ. A γ near one spreads the local reward modifications out to neighboring state/action pairs, increasing the potential optimality loss.

One may question just how much one must alter the reward function to signifi-

cantly impact the learning system. It may appear at first that in order to impact the learning process that adjustments on the order of the average reward would need to be applied. This is not correct because of the *nature* of the applied domain-knowledge-based reward.

In most reinforcement learning systems, reward is parceled out sparingly, with large temporal gaps between reward. Because of discounting, far future rewards have numerically small impacts on the *current* discounted reward function of the system. Domain knowledge, however, is applied continuously. That is, a good domain theory is applicable at all times, and can thus reinforce the learner at every time step. Because domain-knowledge based reward can be given out more often, this reward does not need to be as *numerically* large as the temporally sparse reward given via the original reward function $R(s, a)$.

As a quick example of how much smaller domain-knowledge reward can be, note that, with a discounting factor of $\gamma = 0.9$, a reward of $+12$ at $t = 20$ (twenty time-steps in the future) has the same numerical impact on the $t = 0$ discounted reward function as an immediate, $t = 0$ reward of $+1$.

In fact, a constant infusion of domain-knowledge-based reward can have impacts far beyond remedying problems associated with insufficient training data. For example, the *mixing time* of an MDP refers to the time taken for the average reward expected by an agent to reach within ϵ of its equilibrium value. It has been shown that the computational complexity of reinforcement learning vary polynomially with this mixing time (specifically, proportional to $\frac{1}{1-\gamma}$)[**?**]. A practical way to reduce the mixing time of an RL problem is to reduce the discounting factor γ, though this has the negative practical effect of forcing the learning system to have a more temporally narrow view of reward (that is, far-future rewards have little impact on the current discounted reward when the discounting factor is small). Since domain-knowledge-

based reward is constantly supplied, infusing domain-knowledge into a system can allow the system designer to lower γ since the learning system is receiving more useful 'guiding' reward more often.

D. Fuzzy Reward Shaping

If we accept the dangers faced by reward shaping, the fuzzy knowledge/approximate reinforcement learning platform allows for a very straightforward and concise method for representing knowledge through reward modification as described above. By inspection, one may see that the only place reward enters into the training equations in LSPI is through the term b when solving the system $\mathbf{A}(\pi)\omega = b$. This term is a representation of reward sampled evenly over the state-space of the system:

$$b = \frac{1}{|\mathcal{A}||\mathcal{S}|} \sum_{s \in \mathcal{S}} \sum_{a \in \mathcal{A}} \phi(s,a) R(s,a) \tag{3.21}$$

If we wish to use a post-integration reward function $R'(s,a) = R(s,a) + Ch_F(s,a)$, b becomes:

$$b' = b + b_F \tag{3.22}$$

$$b_F = \frac{C}{|\mathcal{A}||\mathcal{S}|} \sum_{s \in \mathcal{S}} \sum_{a \in \mathcal{A}} \phi(s,a) h_F(s,a) \tag{3.23}$$

As such, a fuzzy rule can be simply represented in the this architecture as a *reward vector*. That is, b_F contains all the information necessary to represent the rule F in the reward function of the system. This form is particularly advantageous since the reward vector only involves h_F and ϕ, which are generally exact, closed form functions, and thus the summation may be able to be done directly in closed form.

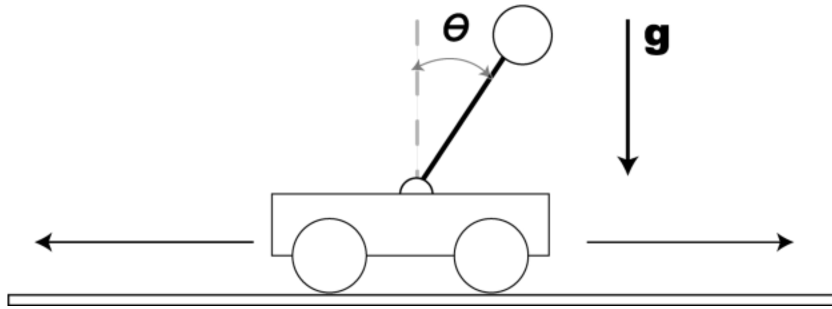

Fig. 8. The cart-pole balancing problem revisited.

E. Integrating Fuzzy Knowledge

It seems realistic that in most situations we will need to integrate not one but *many* fuzzy rules to capture the dynamics of a given system. As such, it would be nice to be able to say that two rules were implemented with equal 'strength', for example. Fortunately, since our fuzzy rules are entirely represented as reward vectors, this type of measured reward distortion is possible. Here we will simply use:

$$b' = C_0 \frac{b}{|b|} + \sum_{i=1}^{N} C_i \frac{b_{F_i}}{|b_{F_i}|} \tag{3.24}$$

where the C_i are weighting constants the relative importance of each reward vector. The reader should note at this point that because of the way LSPI is structured, changes in the magnitude of b have no effect on the final policy (policies are constructed by noting the *relative* differences in reward through the $\arg\max Q(s,a)$). As such, the magnitude of the constants in the above equation are unimportant - only the relative magnitudes can have any effect on the final policy.

F. Cart-Pole Balancing Problem Domain

To demonstrate the integration of domain knowledge using this framework, we chose one of the classic problems in reinforcement learning, the cart-pole balancing problem. In this problem we have a cart of mass M capable of frictionless, one-dimensional motion. Attached to this cart via a hinge is a pole with a mass m on top. Gravity acts to pull the top mass down, but the pole and mass can be kept in the upright position by judicious movements of the cart (see Figure 8 for an account of the quantities involved in this problem).

Agents in this system are capable of applying some impulse either to the left or to the right of the cart, and reward is given out to the agents if the pole is within some angle tolerance of the upright position. System parameters are specified in two variables, θ, the angle of the pole away from the upright position, and $\dot{\theta}$, the angular velocity of the pole. All physical constants and simulation methods (other than one exception noted below) are as described in [23].

As in [23], we collected our training corpora by having an agent follow a random policy (pushing on the cart from the left or the right with equal probability), and observing the reaction of the cart-pole system. In all cases, after the training corpus was collected, we performed LSPI to generate an optimal policy for the balancing agent.

To examine how domain knowledge affected the sample complexity on this domain, we presented the learning agent with various numbers of training examples (from 0 to 20,000), and monitored what percentage of learners, post-training were able to keep the pole upright for at least 100 time-steps. Specifically, we initially generated a corpus of 20,000 training examples, and then sampled subsets of this corpus to obtain however many training examples were required.

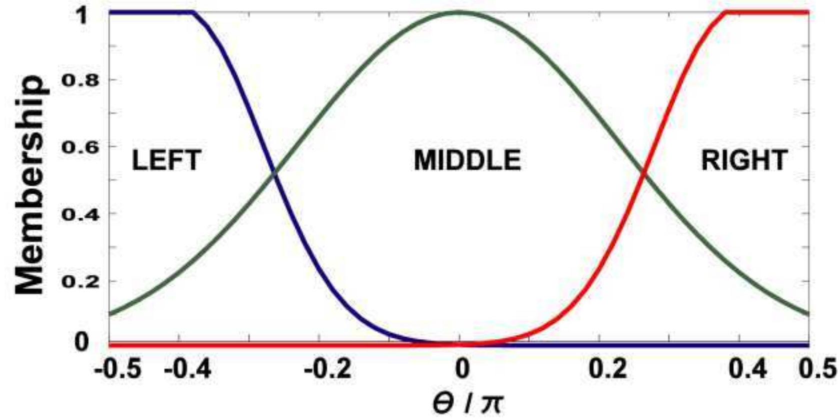

Fig. 9. Fuzzy membership functions illustrated. The membership functions for COUNTERCLOCKWISE, STILL, and CLOCKWISE had the same shape, only they were defined on $\dot{\theta}$ over the domain $\dot{\theta} \in [-1, 1]$.

Table VII. Rules integrated into the inverted pendulum problem

Rule	Description
Rule 1	If LEFT apply force to the right
Rule 2	If RIGHT apply force to the left
Rule 3	If MIDDLE do not apply any force
Rule 4	If CLOCKWISE don't apply force to the right
Rule 5	If COUNTERCLOCKWISE don't apply force to the left

We integrated the fuzzy rules listed in Table VII as outlined in previous sections. The fuzzy membership functions were as Figure 9. We first created a domain-knowledge vector:

$$\mathbf{b_k} = \mathbf{b_{R1}} + \mathbf{b_{R2}} + \mathbf{b_{R3}} + \mathbf{b_{R4}} + \mathbf{b_{R5}} \tag{3.25}$$

We then modified the system reward vector as:

$$\mathbf{b}' = \mathbf{b_0} + C\mathbf{b_k} \tag{3.26}$$

where C was varied between 0 and 1. Look at Figure 3. As we increase the weight of the knowledge vector $\mathbf{b_k}$ the learning agent is able to obtain higher success rates with fewer training examples. Although in all cases the learner exhibits the typical inverse exponential approach to a 100% success rate, the rate at which this is achieved is dramatically increased by the addition of domain knowledge. Also, look at Table II. Here we list the number of training examples where each series intersects the 90% success rate line. Notice that significant performance improvement happens even with only minimal relative scaling of the domain knowledge vector, and that after about $C = 0.1$, further increasing the value of C only slightly decreases the number of training examples required to reach a 90% success rate (Table VIII and Figure 10).

G. Conclusions

We have demonstrated the incorporation of fuzzy domain knowledge into the reinforcement learning process via reward shaping. Specifically, we concentrated on approximate reinforcement learning using the Least-Squares Policy Iteration algorithm

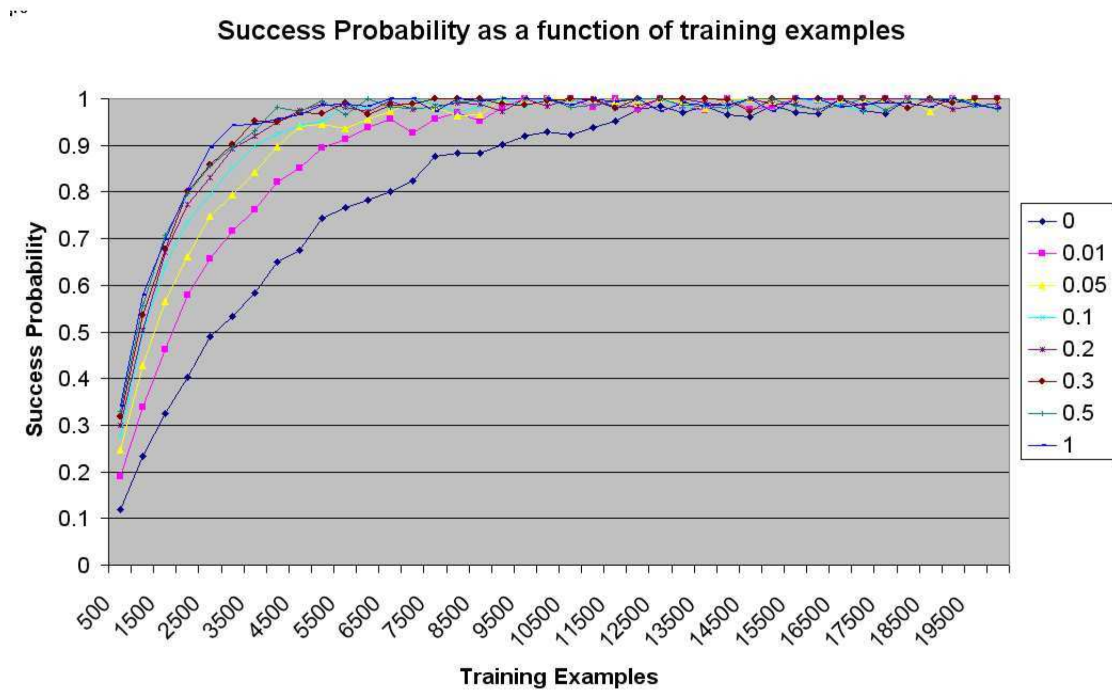

Fig. 10. The results of the sample complexity experiment. As more and more domain knowledge is integrated into the reinforcement learning process, the learner needs fewer and fewer examples to successfully learn how to balance the pole. Different series represent different values of C.

Table VIII. Number of training examples before the learner had a 90% success probability

C	Number of Training Examples (nearest 500 samples)
0	9000
0.01	6000
0.05	4000
0.1	4000
0.2	4000
0.3	3500
0.5	3000
1.0	3000

[23]. Additionally, we derived bounds for how much reward shaping can affect final policy quality.

This method is useful when, as has been our focus in the rest of this dissertation, we do not have complete knowledge of the training domain, and in fact, may only have knowledge enough to write down a few fuzzy rules which govern the domain.

Our simulation domain consisted of a training-example-limited version of the cart-pole balancing problem such that without external domain knowledge, standard LSPI reinforcement learning produced a poorly-performing controller when insufficient training examples were supplied to the learner. We encoded domain knowledge in simple rules that one would reasonably expect a pole-balancing controller to follow and integrated the knowledge into the learning environment via reward shaping as outlined in this chapter. With domain knowledge integration, we saw that the learner was able to achieve higher success rates with fewer training examples. Finally, we saw how even slightly modifying the reward vector could have significantly decrease

the sample complexity of the learning process.

CHAPTER IV

LEAST-SQUARES POLICY ITERATION OVER CONTINUOUS ACTION SPACES

A. Introduction

In some domains, using a continuous action space instead of a discrete action space can result in computational savings. The main problem we have addressed throughout this dissertation is how we can minimize the negative effects of exponential action-space scaling in multi-agent systems. In domains where continuous action-spaces are needed (consider any agent embodied in an analog controller, for example) one can often either use a single continuous action (say, $a = [-10, +10]$), or create many actions corresponding to gradations of the action's desired range (i.e, $a_1 = -10, a_2 = -9, a_3 = -8, \ldots, a_{N-2} = 8, a_{N-1} = 9, a_N = 10$). Even worse, imagine if we need to train *many* agents with such action spaces. In the first example, with a single continuous action space, our multi-agent action space would simply scale as $O(C^n)$ (where n is the number of system agents), whereas in our discrete gradation-based approach, the action space would scale as $O((10C)^n)$.

As in previous chapters, here we will focus on the Least-Squares Policy Iteration algorithm [23]. Interestingly enough, LSPI is already theoretically able to learn over continuous action spaces (that is, nothing about the algorithm's construction assumes that the action spaces are discrete). However, we show in this chapter that there are several significant practical implementation issues that may arise when one tries to use LSPI with continuous action spaces in certain domains.

First, we show how symmetry effects in the basis functions used to approximate

the discounted reward function Q can cause the typical action-selection mechanism used in discrete-action LSPI to fail on "narrow-Q" domains (commonly encountered domains where Q-values are numerically very close to one another due to high discounting factors, system structure, etc.). We provide a simple, theoretically justified heuristic solution to this action-selection problem and show how it is possible to successfully train high-performance, continuous-action controllers using LSPI under the heuristic. We evaluate the performance of these controllers on a noisy inverted pendulum domain where a weighted pole must be kept upright in the presence of strong, random, externally applied 'noise' forces. In this domain, the controller must be able to apply gentle, corrective forces in the absence of wind, and much stronger corrective forces when wind is present. As such, this domain is well-suited to a continuous-action (here continuous-force) controller. Finally, we examine some practical issues relating to the speed with which a trained continuous-action controller can operate (since the number of actions is no longer finite, controller operation/action selection necessarily involves a maximization search). We provide a framework for using stochastic search methods to find Q-maximizing actions and show how one can exploit the temporal coherence of the controller response to speed up action selection.

B. Related Work

Eventually, the simulation domains we work with in this chapter will be fairly simple. In fact, we will work with an enhanced pole balancing simulator. This is worthy of note because inherently, the experimental domain is not difficult in general, and in fact there exist a myriad of techniques to directly solve the pole balancing problem in many domains. The dynamics of the problem are exactly known, so that a closed-

form controller to perfectly keep the pole upright is possible. Additionally, it is a simple matter to develop a fuzzy-logic based controller for this problem, and various approaches such as genetic algorithms [96] and direct hand-coding have been used.

In this chapter we examine a specific problem that occurs specifically when using LSPI on continuous action spaces. This problem is not inherent to our simulation domain, and in general it does not plague other machine learning techniques which might be used to solve this problem. The problem we will introduce occurs specifically in reinforcement learning because of the way the algorithm chooses to represent reward. As we will see, by discounting reward through time, the algorithm opens itself up to action-selection failure on certain domains. This line of research is not at all relevant to other learning algorithms which do not employ reward discounting.

Thus, it is not the experimental domain itself that is useful, but rather the demonstration of reinforcement learning on the domain. Other techniques could solve the 'problem' easily, and without the problems that are incurred by using RL. Again, our experimental domain is only a successful demonstration of our technique. There are certainly domains where the type of continuous-action-space RL would *need* to be used to the exclusion of other techniques.

C. Continuous-Action Selection Failure on Narrow-Q Domains

1. Basis Functions

Most published LSPI work done thus far has been done with discrete action spaces such that basis functions of the following form are used:

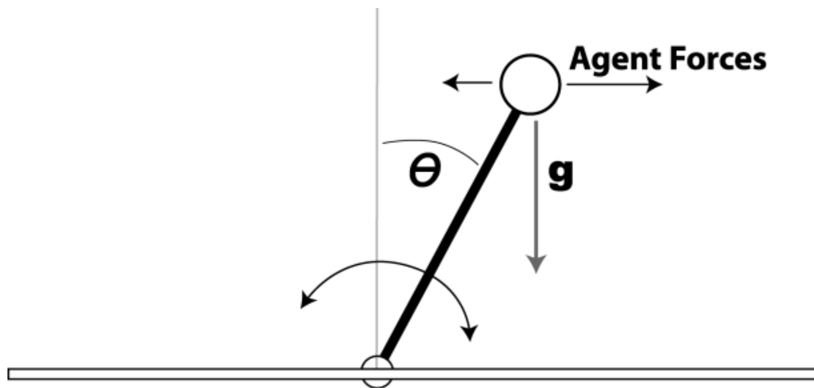

Fig. 11. The inverted pendulum balancing problem.

$$\phi_i(s, a) = \psi_i(s)\delta_{a,a'} \tag{4.1}$$

for some a' indexed by i. That is, there is generally some state-related part of the function (for example, radial basis functions are often used), and then there is an action-part which has generally been a delta function: if a is equal to some action a' then the basis function is simply the state-based part ψ_i, otherwise it is zero. For example, in the typical LSPI pole balancing problem, the system state is measured by the pole's angle with the vertical θ and the pole's angular velocity $\dot{\theta}$ - additionally, the system has access to three actions, impart an impulse to the left or right, or impart no impulse (these quantities are illustrated in Figure 11). If we define the following vectors then:

$$\mathbf{x} = \{-\frac{\pi}{4}, 0, \frac{\pi}{4}\}, \mathbf{y} = \{-1, 0, 1\} \tag{4.2}$$

$$\mathbf{z} = \{\text{LEFT,NONE,RIGHT}\} \tag{4.3}$$

then reasonable basis functions would be [23]:

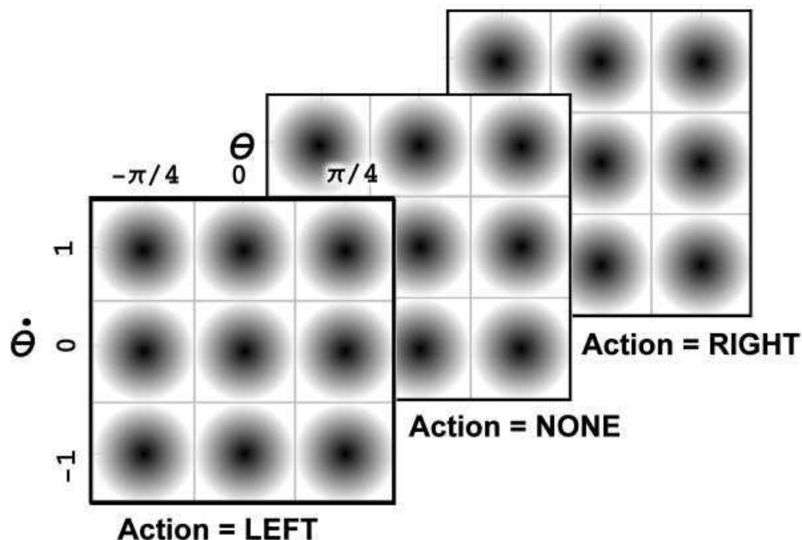

Fig. 12. Illustration of the basis functions used in the IP problem.

$$\phi_{m,n,p}(s = \{\theta, \dot{\theta}\}, a) = e^{-((\mathbf{x}_m - \theta)^2 + (\mathbf{y}_n - \dot{\theta})^2)/2} \delta_{a,\mathbf{z_p}} \tag{4.4}$$

for a total of 27 basis functions illustrated in Figure 12. Generally, these functions are indexed through a single index i (here $1 \leq i \leq 27$). If one wishes to use LSPI in the context of continuous-action spaces, the discrete delta-function action-dependence of the basis functions must be replaced with some sort of continuous dependence. Although this could be accomplished in such a way such that the basis functions ϕ_i were no longer sparable into a state-based and action-based part, it has generally been a common practice to make the basis functions separable even over different state space dimensions (note that the $\phi_{m,n,p}$ above is simply a basis function in θ multiplied by a basis function in $\dot{\theta}$), we will proceed by assuming that our basis functions are of the form:

$$\phi_i(s, a) = \psi_j(s) \chi_k(a) \tag{4.5}$$

2. Symmetry-Induced Problems with Action Selection

In policy iteration, after one has found the Q-values of the previous policy, an improved policy is constructed as:

$$\pi'(s) = \arg\max_a \hat{Q}(s,a) \tag{4.6}$$

$\hat{Q}$ in the LSPI framework is of course just:

$$\hat{Q}(s,a) = \sum_i \phi_i(s,a)\omega_i = \sum_j \sum_k \psi_j(s)\chi_k(a)\omega_{j,k} \tag{4.7}$$

$$= \sum_k \chi_k(a) \sum_j \psi_j(s)\omega_{j,k} = \sum_k \chi_k(a)\hat{Q}_k(s) \tag{4.8}$$

The reader may observe that in the case of discrete action spaces, the $\hat{Q}_k(s)$ are simply the expected discounted rewards of choosing action a_k in state s (because in the case of discrete actions $\chi_k(a) = \delta_{a,a_k}$). As such, in some domains the $\hat{Q}_k(s)$ may be numerically very similar, and we call these domains *narrow-Q domains*:

$$\forall k, k' \quad \left|\hat{Q}_k(s) - \hat{Q}_{k'}(s)\right| \ll \hat{Q}_k(s) \tag{4.9}$$

3. A Practical Example of an Extreme Narrow-$\hat{Q}$ Domain

The reader should note that examples of domains where narrow-Q effects manifest significantly are not that far fetched. One potential situation where this might happen is a learning environment with a high discounting factor $\gamma \approx 1$: For example, consider the case of the inverted pendulum balancing problem where positive reward is given when the pendulum is within some angle-tolerance of upright and negative reward is

given when the pendulum is outside this zone (fallen over). Now imagine that the domain is such that even the best controller can hold the pendulum up for no more than K time-steps (perhaps there are random forces applied to the system). Finally, imagine that we have set the pendulum in the upright position and that we attempt to estimate the average discounted reward obtained if we apply a force to the left, to the right, or apply no force at all. Since no matter which action is performed the system will not be able to keep the pendulum upright for more than K time-steps, the discounted reward for all three actions will look something like this:

$$\hat{Q}(s = \text{upright}, a) = C_a + \sum_{t=K+1}^{\infty} \gamma^t * (-1) = C_a - \frac{\gamma^{K+1}}{1-\gamma} \tag{4.10}$$

$$\lim_{\gamma \to \infty} \left(C_a - \frac{\gamma^{K+1}}{1-\gamma} \right) / \left(C_{a'} - \frac{\gamma^{K+1}}{1-\gamma} \right) = 1 \tag{4.11}$$

such that the expected rewards of the three actions can be made relatively infinitesimally close together by taking γ increasingly closer to 1.

4. Action Selection: The Problem

Having $\hat{Q}_k$ relatively very similar can lead to the following problem when selecting $\hat{Q}$-maximizing actions over continuous action spaces: Imagine that we have a one-dimensional action space defined by the action variable a, and we choose to represent this space in terms of radial basis functions (a not implausible choice considering the popularity of this basis for representing state spaces for LSPI). Further (as is typical), imagine that we lay our basis functions evenly spaced along the a dimension. Specifically, let us say that we use three basis functions $\chi_{-1}, \chi_0, \chi_1$ (illustrated in Figure 13C):

$$(example) \quad \chi_i(a) = e^{-(a-3i)^2/2} \tag{4.12}$$

such that our basis functions are simple Gaussians centered at $-3, 0, 3$ respectively. Now imagine that we work in a narrow-Q domain such that all three $\hat{Q}_k(s)$ are roughly the same, so that effectively:

$$\hat{Q}(s,a) \approx \hat{Q}_1(s) \sum_k \chi_k(a) \tag{4.13}$$

Thus, in this case, the $\hat{Q}$-maximizing action never changes as we move from state to state (the state-dependent scaling $\hat{Q}_1(s)$ factor cannot change which action is $\hat{Q}$-maximizing), and as such, the *system state* no longer determines the optimal action. Instead, this is determined entirely by *symmetry effects* in the action-space basis functions χ_k. For example, here the χ_k functions collectively reinforce the $a = 0$ region such that $\hat{Q}(s, a = 0)$ is larger than any other point in the action space. In this case then, the optimal policy would perform $a = 0$ in every state. Now, in general most domains will not be such that all $\hat{Q}_k$ are identical, but there is still the danger of symmetry-based effects in the action-basis functions overwhelming the effects of state-dependent $\hat{Q}_k(s)$ weighting. One could design around this by choosing a set of basis functions where this did not occur, but the problem is that the narrowness of the $\hat{Q}_k(s)$ is dependent on the current state of the system. In some states the $\hat{Q}_k$ may be vastly different and diminish the action-basis symmetry effects, and in other states, all the $\hat{Q}_k$ may very well be identical. This uncertainty makes it very difficult to, with certainty, design one's way around such symmetry-related problems. The reader should also note that symmetry effects are only heightened in multi-dimensional action spaces. In Figure 13C we have illustrated a two-dimensional analogue of the one dimension

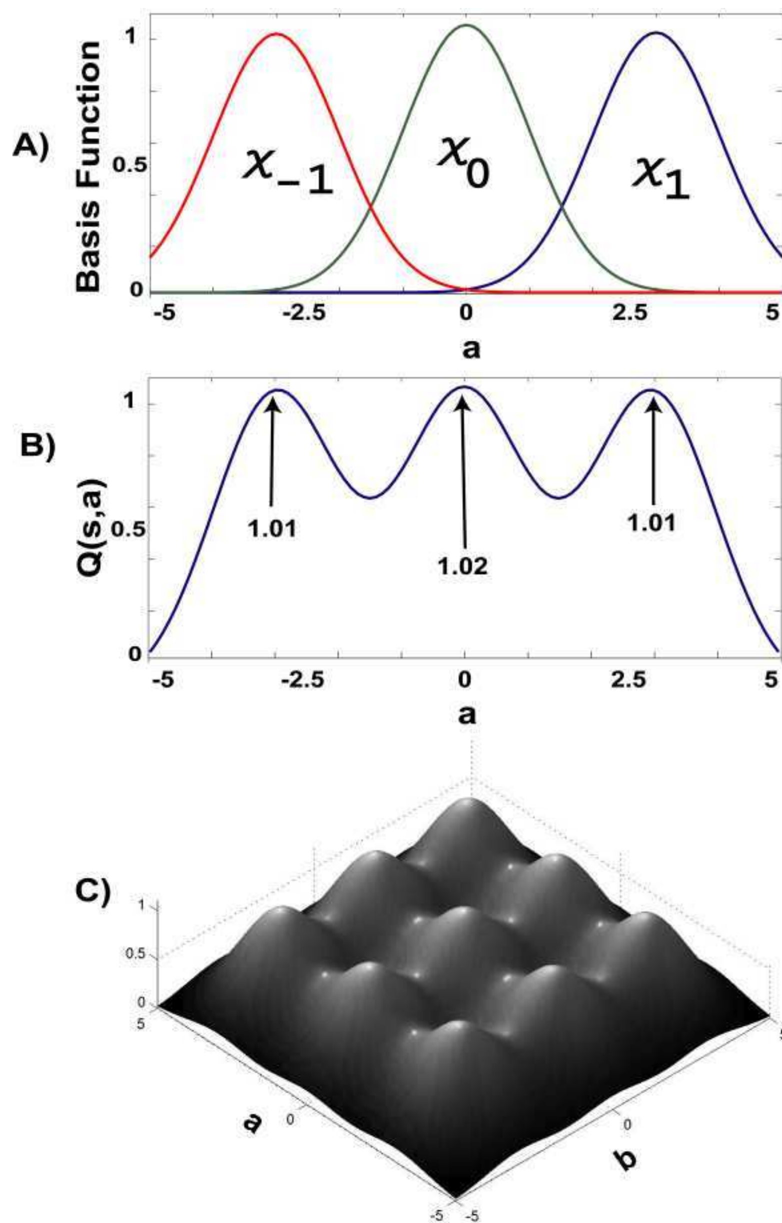

Fig. 13. Illustration of the Narrow-Q problem. (A) Illustrates sample basis functions discussed in this section (B) Illustrates the $\hat{Q}(s,a)$ function *for all states* in a narrow-Q domain where the $\hat{Q}_k$ are very similar (C) Illustrates a two-dimensional version of the same situation.

example presented here - the reader may verify that in this case that the outside 8 basis functions contribute to make the origin the highest-valued point in the state space.

5. Solutions

Three possible solutions present themselves for addressing symmetry-effects in narrow-Q domains:

- Try to increase the numerical variance of the $\hat{Q}_k$ by lowering the discounting factor γ.
- Decrease the spread of the basis functions such that symmetry effects *never* manifest.
- Forcibly make the domain wide-Q by normalizing the $\hat{Q}_k$ prior to action selection.

The first option is unnaceptable because it changes how the learner values future rewards relative to immediate rewards, and additionally, there is no *guarantee* that this would result in increasing the spread of the $\hat{Q}_k$ (it could work in the pendulum from earlier, but this is not a general result). The second option would work, but would also eliminate much of the apppeal of working in continuous action spaces. That is, the overlapping nature of the action basis functions is what makes learning on continuous action spaces interesting in the first place (if the action-basis functions didn't interact at all, we would be left with a discrete-action system). Thus, we will concentrate on the last option - forcing our domain to be less 'narrow' by rescaling the $\hat{Q}_k$ prior to action selection.

6. Heuristic for Transforming Narrow-Q Domains

We propose a suitable heuristic method for avoiding the problems of overwhelming symmetry effects on narrow-Q domains described in the last section: Instead of performing action selection on $Q(s, a)$ directly, we perform action selection on $Q'(s, a)$ such that:

$$Q'(s,a) = \sum_k \chi_k(a) \left(Q_k(s) - A\right) / B \tag{4.14}$$

$$A = \min_k Q_k(s), \quad B = \max_k Q_k(s) - A \tag{4.15}$$

This normalizes the $\hat{Q}_k$ such that the largest $\hat{Q}_k$ is scaled to 1 while the smallest is scaled to 0. Thus, we have a guaranteed range for our $\hat{Q}_k$ (for all states), and as such it becomes possible to design action-basis functions such that there is a pre-determined contribution from symmetry effects.

However, obviously we cannot arbitrarily change the $Q(s, a)$ function without repercussions. Fortunately, it is possible to determine the worst-case effects of this transformation.

Theorem 1: If we normalize a system's Q-function as:

$$Q'(s,a) = \sum_k \chi_k(a) \left(Q_k(s) - A\right) / B \tag{4.16}$$

$$A = \min_k Q_k(s), \quad B = \max_k Q_k(s) - A \tag{4.17}$$

then at worst, deviation in system performance can be bounded by:

$$\lim_{m\to\infty} \sup ||\hat{Q}^{\hat{\pi}'_m} - Q^*||_\infty \leq \frac{2\gamma(\epsilon_{approx} + \epsilon_{basis})}{(1-\gamma)^2} \tag{4.18}$$

$$\epsilon_{basis} = \left|\max_{s,a} A\right| \left|\max_a \left(\sum_k \chi_k(a) - \min_a \sum_k \chi_k(a)\right)\right| \tag{4.19}$$

Where π'_m is the policy resulting from action selection on the normalized Q-function Q', and $\epsilon_{approximation}$ is any error introduced strictly from using basis functions to approximate the Q-function.

Proof: First of all, note that the only thing that we will use this modified Q' for after the modification is to perform action selection. As such, we have two degrees of freedom in the *reward* function to work with. That is, if I maximize a system's $Q(s, a)$ function in a given state s and get an action a, then if I alter the reward function $R(s, a)$ for the system either by a *uniform translation* or an *overall scale transformation*, $Q(s, a)$ should still be maximized by choosing a in state s. That is, *for the purpose of action selection*:

$$R(s, a) \equiv B(R(s, a) + A) \tag{4.20}$$

This invariance in the reward function R carries over into the same type of invariance in the Q function (if I want to scale Q by B then I just scale R by B, and if I want to add A to $Q(s, a)$ for all s and a, then I just add $(A - 1)/A$ to $R(s, a)$ for all states and actions. This is not a new observation made here (see [95] for a very detailed derivation and analysis of the implications).

As such the overall scale transformation B will be irrelevant to us, since we are always free to perform a scale transformation on $Q(s, a)$ *without* repercussions (no

different than scaling all rewards in the system). The A factor is what we must consider. Specifically,

$$Q'(s,a) = Q(s,a) + \sum_k \chi_k(a) A \tag{4.21}$$

Which is not uniform, but has action-dependence component. However, if there were some constant component to this expression, we could just subtract it out without affecting action selection. We can rewrite the above expression as:

$$Q'(s,a) = Q(s,a) + A\left(\min_a \sum_k \chi_k(a)\right) + A\left(\sum_k \chi_k(a) - \min_a \sum_k \chi_k(a)\right) \tag{4.22}$$

Now, the middle term in the above expression is not constant, because A can vary between states. However, because this is a narrow-Q domain, we will make the approximation that essentially, $\forall s, A(s) = A_0$. If we approximate the middle term as constant, we can proceed to ignore it for the purposes of action selection:

$$Q'(s,a) = Q(s,a) + A\left(\sum_k \chi_k(a) - \min_a \sum_k \chi_k(a)\right) \tag{4.23}$$

or:

$$Q'(s,a) = Q(s,a) + \epsilon \tag{4.24}$$

$$\epsilon_{basis} = A\left(\sum_k \chi_k(a) - \min_a \sum_k \chi_k(a)\right) \tag{4.25}$$

but this is bounded because our basis functions are finite by assumption:

$$|\epsilon_{basis}| < \left|\max_{s,a} A\right| \left|\max_{a}\left(\sum_{k}\chi_k(a) - \min_{a}\sum_{k}\chi_k(a)\right)\right| \tag{4.26}$$

However, because in this context we deal with approximate policy iteration (the Least-Squares Policy Iteration algorithm), we are given some lee-way on how much we distort the Q function. In fact, if we remember back to the Bertsekas theorem used in Chapter IV, where if we could guarantee that (at each iteration):

$$||\hat{Q}^{\pi} - Q^{\pi}|| < \epsilon \tag{4.27}$$

then we could guarantee that our final policy would be such that:

$$\limsup_{m\to\infty} ||\hat{Q}^{\hat{\pi}_m} - Q^{*}||_{\infty} \leq \frac{2\gamma\epsilon}{(1-\gamma)^2} \tag{4.28}$$

In this case, in addition to whatever approximation error there is already, we will add

$$\epsilon_{basis} = \left|\max_{s,a} A\right| \left|\max_{a}\left(\sum_{k}\chi_k(a) - \min_{a}\sum_{k}\chi_k(a)\right)\right| \tag{4.29}$$

error and will have:

$$\limsup_{m\to\infty} ||\hat{Q}^{\hat{\pi}'_m} - Q^{*}||_{\infty} \leq \frac{2\gamma(\epsilon_{approx} + \epsilon_{basis})}{(1-\gamma)^2} \tag{4.30}$$

where π'_m is the policy resulting from action selection on the normalized function Q'.

■

7. Reducing the Heuristic Error

Let us briefly examine the structure of ϵ_{basis} and look at ways to reduce this quantity. The reader should notice that second term in ϵ is tied to how much 'hilliness' is in the basis functions. That is, in the extreme case of a set of constant basis functions, this quantity would be zero, and in the case of using Dirac delta functions as basis functions, this quantity would be quite high. Figure 14 actually calculates some approximation errors for simple radial basis functions. Thus we see that while this heuristic does introduce some error into the approximation process, the magnitude of this error can be reduced by choosing basis functions which adequately cover action space.

D. Experiment 1: Continuous Pole Balancing

To illustrate the use of our normalization technique on continuous action-spaces, we chose the problem referenced earlier in this paper, the inverted pendulum domain. In this domain we have a pendulum of length L with a weight of mass M attached to the top of it. The pendulum is attached to the ground via a hinge, and gravity works to continually pull the mass off-center.

Agents in this system are capable of applying some impulse either to the left or to the right of the pendulum, and reward is given out to the agents if the mass is within some angle tolerance of the upright position. System parameters are specified in two variables, θ, the angle of the mass away from the upright position, and $\dot{\theta}$, the angular velocity of the mass. All physical constants and simulation methods (other than one exception noted below) are as described in the presentation of the inverted-pendulum domain in [23]. Generally, this problem is well-solvable with discrete-action spaces, so

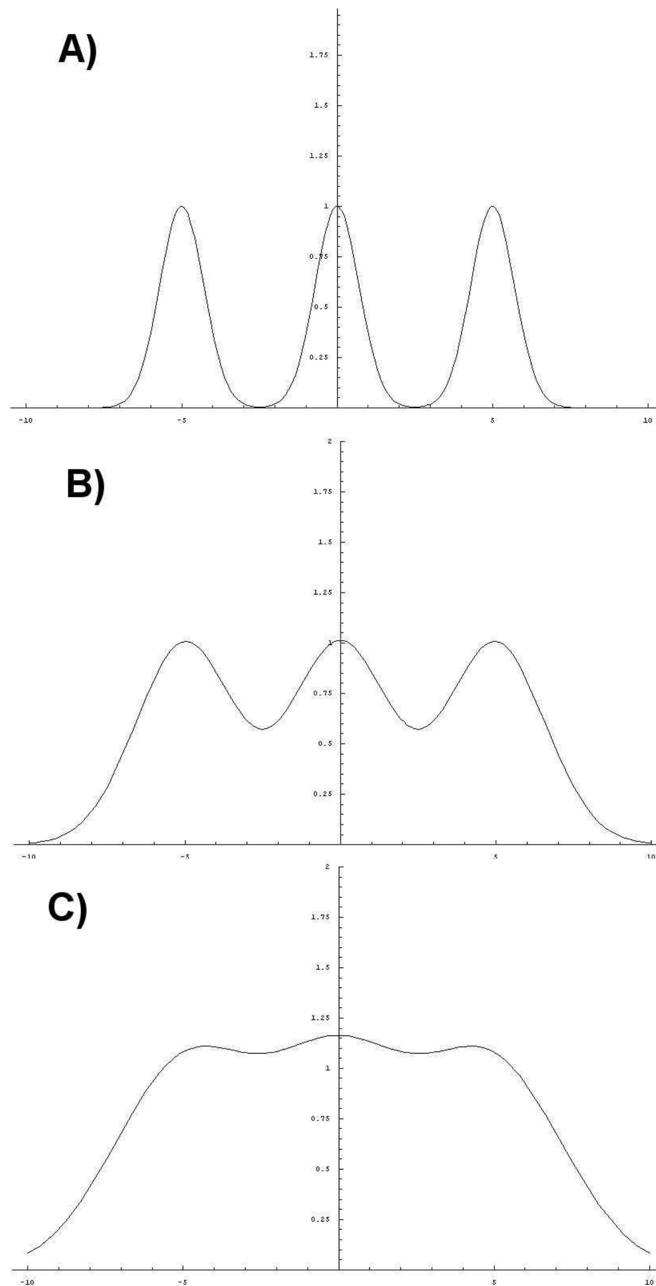

Fig. 14. Approximation error and various basis functions. Three different sets of basis functions. Each plots $\chi_k = e^{-(a-c_k)/b}$ where $c_k \in \{-5, 0, 5\}$. A) plots $b = 1$, and has a $|\epsilon_{approx}| < 0.99$, B) plots $b = 5$ and has a $|\epsilon_{approx}| < 0.43$, and C) plots $b = 10$ and has a $|\epsilon_{approx}| < 0.03$. Notice how there is less approximation error as the basis functions fill more of the space.

to better illustrate the use of our methods (the use of continuous actions spaces), we modified the domain slightly: in addition to gravity, our mass is subject to a 'wind' force that applies occasional, random force to the mass that is roughly an order of magnitude stronger than the forces applied by gravity. Specifically, we found that the forces exerted by gravity were on the order of ($\pm$30N) and we allowed our random 'wind' forces to assume the random ($\pm$300N).

The reason for the modification is as follows: Generally, in the discrete domain, the agent is able to supply a *fixed* force from either the right or left, or choose not to apply a force. In situations where the mass is very near vertical, only small forces need to be occasionally applied to correct the position of the mass as it drifts off-center due to gravity. Empirically, it is found that the system is easily made unstable by giving the agent the ability to apply too much force. That is, if the agent's actuators are set very high (in comparison to the rest of the physical constants in the simulation), choosing to impart a force in a given direction has significant consequences for the dynamics of the system (since each action imparts a large amount of energy), and as such the controller has a harder time keeping the mass centered. Generally this is not an issue since only small corrective forces need be applied and the maximal agent force can be capped. However, in the case of a strong-noise-wind domain, the agent *must* be able to apply significant forces to counteract the effect of the wind. Thus a successful controller must operate well in two domains. On the one hand the controller must be able to impart a great deal of force very suddenly to oppose any strong-wind forces when they occur, but on the other be able to apply gentle, corrective-only forces to the pendulum when the system is near-upright to maintain stability. Thus, this strong-wind-noise domain is well-suited to illustrating the advantages of a controller capable of applying continuous levels of force (continuous action spaces).

We simulated several different versions of this problem to demonstrate our nor-

malization technique. Specifically, we simulated the problem with and without the random strong forces with several different 'builds' of agents. We began with the standard pole-balancing agent which was able to apply a fixed amount of force in either direction. We then implemented an agent which was able to apply multiple force levels in either direction, and finally, an agent which was actually able to apply continuous amounts of force. For our continuous action basis functions (necessary to train an agent to operate over a continuous action space) we used two sets of functions. First we used the radial basis functions that we have used throughout the rest of this dissertation. Additionally, we used the first few terms of the Fourier expansion. Specifically, our second set of action basis functions were of the form:

$$a(x) = \frac{1 + cos(C\pi x)}{2} \tag{4.31}$$

$$b(x) = \frac{1 + sin(C\pi x)}{2} \tag{4.32}$$

$$c(x) = \frac{1 + cos(2C\pi x)}{2} \tag{4.33}$$

and

$$\chi(\theta, \dot{\theta})_{q \in \{a,b,c\}, t \in \{a,b,c\}} = q(\theta)t(\dot{\theta}) \tag{4.34}$$

where the Cs were chosen to normalize the data (for both θ and $\dot{\theta}$ to between -1 and 1. This choice of basis functions still allows us to adequately express functions that are spatially localized and demonstrate our technique with a basis set other than radial basis functions. In total, we simulated seven different agent configurations as listed

Table IX. Agent simulation settings

Agent Description	Action Space	Forces?
2-action, weak	{-30N, 0, +30N}	No
2-action, weak	{-30N, 0, +30N}	Yes
2-action, strong	{-300N, 0, +300N}	Yes
5-action, strong	{-300N, -150N, 0, +150N, +300N}	Yes
7-action, strong	{-300N, -200N,-100N, 0, +100N, +200N, +300N}	Yes
cont., w/ norm.	[-300N,+300N]	Yes
cont., w/o norm.	[-300N,+300N]	Yes

in Table IX, in each case performing training separately under two different sets of basis functions (RBFs and Fourier functions).

In the case of the continuous controller, action selection (Q-maximization) was performed by gridding up the function into $.05N$ increments and performing a brute-force search of the state space.

1. Results

The results of this experiment are displayed in Figure 15. As expected, in the base agent simulation configuration, where there are no additional strong random 'wind' forces, the weak agent is able to keep the pole up indefinitely. The basic pole balancing problem is not particularly difficult, and the agent, while weak, could apply more than enough force (±30N) to overcome the gravitational pull on the pole.

Next we tested the same agent in the presence of added, strong, random forces (up to $300N$ in either direction). Not surprisingly, the agent was unable to keep the pole up for any significant amount of time. In fact, the agent was generally able to

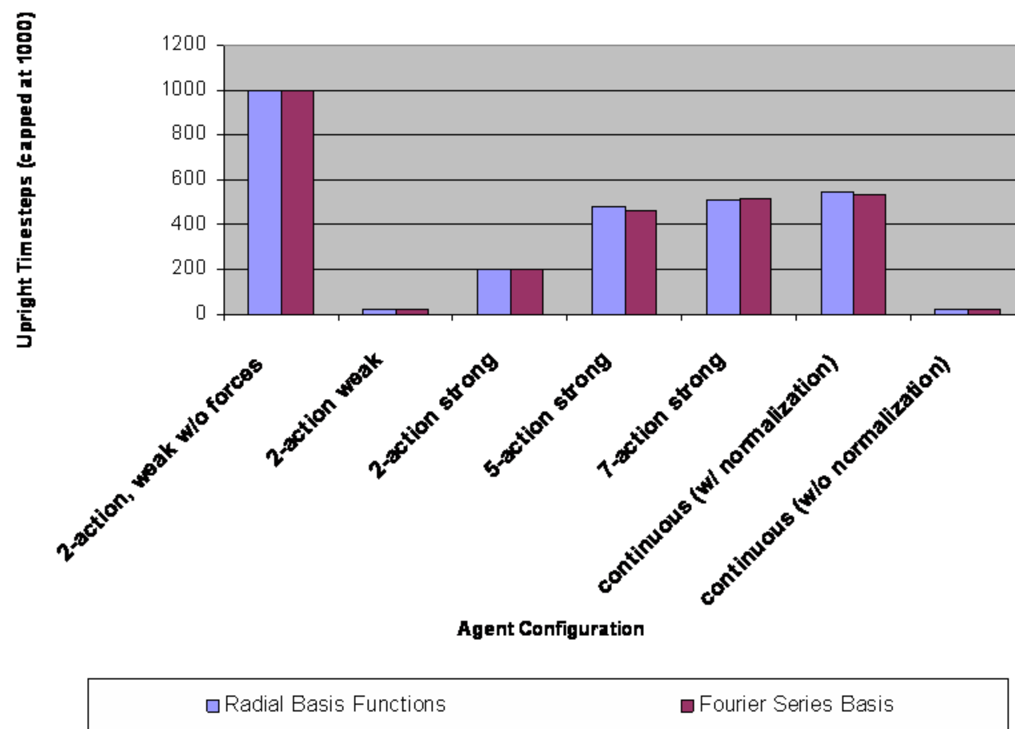

Fig. 15. Results for the “Experiment 1” set of experiments.

keep the pole up only for as long as it took the pole to fall over in the simulation. This failure is due entirely to the fact that the agent was simply unable to apply enough force to counteract the $300N$ magnitude forces being injected into the simulation.

Next we tested the agent which was able to apply $300N$ in either direction. This agent, by design, was able to supply enough force to counteract the strong random forces being injected into the system, but looking at the results of the experiment, we see that the agent did not perform very well. The reason is that, although the agent was able to counteract the strong injected forces, for much of the simulation, these forces were absent, and the agent merely had to contend with small gravitational forces. Being only able to exert large amounts of force, the agent, in trying to make small corrective adjustments to the position of the pole, made the system unstable. So, while the pole did not immediately fall over because of the injected random strong forces, the agent was not able to keep the system upright for any length of time.

To address this, we implemented a 5-action agent that was able to apply two levels of force in either direction ($300N$ and $150N$). This agent did much better, presumably because it was able to supply large corrective forces when large injected random forces were present, but also to use less intense forces to keep the system upright when only gravity was acting on the system.

We then took this further, and implemented an agent capable of applying three levels of force in either direction ($300N$,$200N$, $100N$). Not surprisingly the agent performed even better. This increase in performance did come with an increase in the size of the action space of the system (which was not computationally prohibitive since we only dealt with a single agent).

Finally, we implemented an agent with a fully continuous action space. While using an action space that was *smaller* (1 continuous action) than the previous four agent simulation settings (2,5, and 7 actions), the system was able to keep the pole

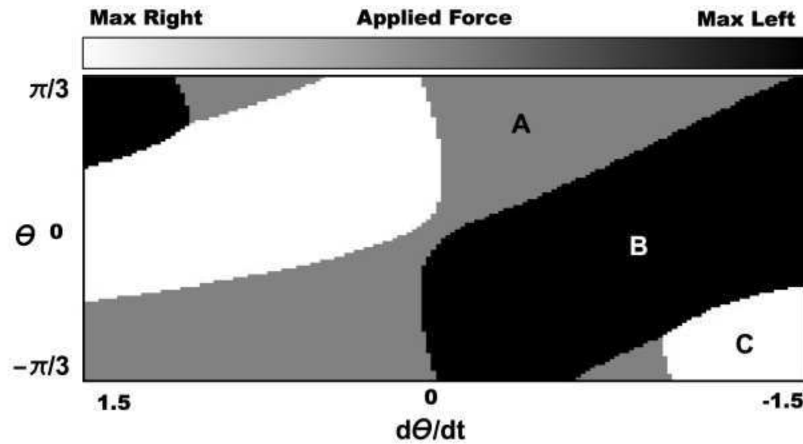

Fig. 16. Force response diagram for the 2-action strong agent.

upright for *longer*. We experimented with this agent both with and without Q normalization (as described earlier in this chapter). With normalization, the system performed well ... without normalization, the controller suffered from exactly the kind of basis-function symmetry generated problems we described earlier in this chapter. Without normalization, in all attempts the system (weight vector ω) failed to converge to a useful policy during policy iteration. Upon investigation, we found that using the standard action selection rule consistently resulted in choosing a zero force in the case of the radial basis function action basis functions and resulted in choosing a slightly positive constant force in the case of the Fourier series action basis functions (the sum of the first three terms of the Fourier expansion has a maximum slightly to the positive side of the origin).

In Figure 16 we plot the force zones for the continuous-action controller (the discrete action zones are plotted in Figure 17). Notice that it has much the same form as the discrete-action policy, with the major difference being that the zone B actions which drive the system into the corrective corner zones C are much less harsh (these regions apply significantly less force than the corner corrective zones). In fact,

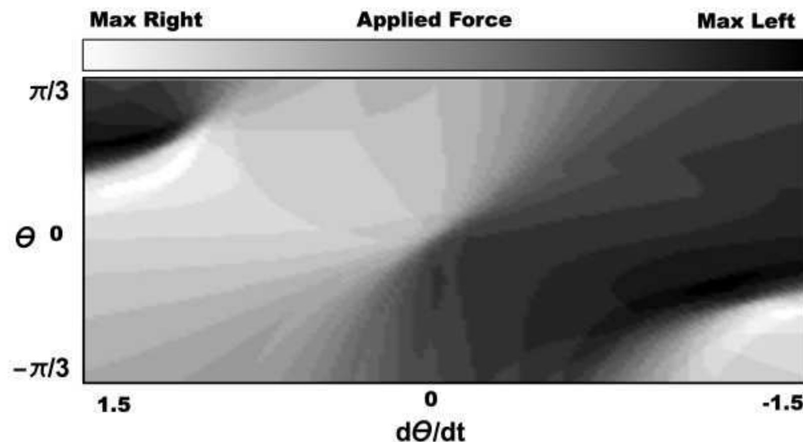

Fig. 17. Force response diagram for the continuous-action agent.

for small deviations from upright, the system seems to gently guide the system into the corner corrective zones, and when in those zones, apply a large corrective force.

In all cases we trained the agent using radial basis functions and the Fourier-type basis functions. No significant performance differences were seen between the two sets of basis functions. This would suggest that the benefits of normalization we saw here are fundamental and not a side-effect of the particular basis functions we used (see Table X).

E. Stochastic Action Selection

Note that in continuous action spaces, finding the Q-maximizing action can no longer be done for general systems by simply enumerating the possible actions and then choosing the one with the highest Q (we could use this approach for our domain, but in multi-dimensional action spaces, this approach would become too expensive). Rather $Q(s, a)$ becomes a continuous function in a and search techniques must be used for maximization. Another issue to consider is that LSPI is commonly used to train

Table X. Upright timesteps (capped at 1000) for the various simulation environments (one STD shown)

	Upright TimeSteps	Upright TimeSteps
Method	Radial Basis Functions	Fourier Series Basis
2-action, weak, w/o forces	1000	1000
2-action, weak w/ forces	25 $\pm$ 14	23 $\pm$ 16
2-action, strong	202 $\pm$ 49	197 $\pm$ 51
5-action, weak	480 $\pm$ 125	462 $\pm$ 115
7-action, weak	510 $\pm$ 131	518 $\pm$ 119
continuous action, w/ norm.	550 $\pm$ 125	531 $\pm$ 122
continuous action, w/o norm.	26 $\pm$ 20	28 $\pm$ 19

system *controllers* (since it can operate over continuous state spaces) - e.g., pendulum balancing and bicycle riding [23]. As such, it it often useful to be able to have the controller select appropriate actions in a timely manner. Thus, ideally we would like some mechanism to find Q-maximizing actions over continuous Q functions quickly. Gradient ascent might be an appropriate option here (for speed), but in general one expects the Q function to be rather hilly (consider even the simple examples we have analyzed in this paper). As such our approach will be to use stochastic search (specifically simulated annealing) to perform action selection. After providing a framework for doing this, we will show how the explotation of temporal-coherence in the controller's responses can be used to speed up this stochastic search.

1. Simulated Annealing-Based Action Selection

Simulated annealing [97, 98, 99] is a physically-inspired search algorithm which mimics how a metal acquires lowest-energy configurations by gradual temperature reduction [97]. There is exhaustive documentation on how to use the algorithm in various domains, but briefly, the algorithm seeks to maximize an objective function $f(\mathbf{x})$ through psuedo-random state space exploration. Updates to the current state are accepted directly if they increase the objective function f (if $f(\mathbf{x} + \Delta\mathbf{x}) > f(\mathbf{x})$ in other words) and accepted with probability $e^{-(f(\mathbf{x}) - f(\mathbf{x}+\Delta\mathbf{x}))/T}$ otherwise, where T is a temperature parameter which is decreased according to some *cooling schedule*.

To use simulated annealing to perform action selection (in a given state) in our framework, one simply computes the set of $Q_k(s)$ and then attempts to maximize the function:

$$Q'(s,a) = \sum_k \chi_k(a) \left(Q_k(s) - A\right) / B \tag{4.35}$$

as before. Termination of the algorithm is generally set for when no improvement of the objective function has been seen for some number N search steps.

2. Experiment 2: Simulated Annealing Action Selection

To see how well stochastic action selection worked in our inverted pendulum domain, we performed Experiment 1D of the previous section using simulated annealing for action selection. We used a naive cooling schedule of $T = 1/\sqrt{i}$ where i is the step-number. No particular attempt was made to optimize the form of the cooling, which is to say it is likely performance enhancements could be obtained over the reported results by some careful modification of the cooling schedule [100, 101, 102]. Steps were

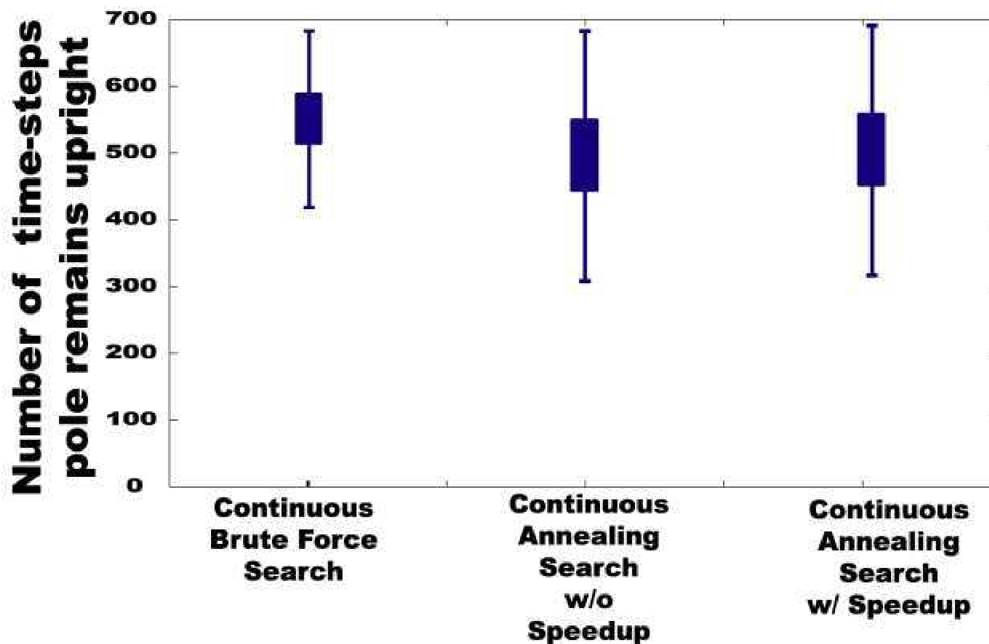

Fig. 18. Results for the "Experiment 2" set of experiments.

generated as $\Delta\mathbf{x}_i \in [-1, +1]$ for all search steps, and this is also a naive approach, as generally one wants to use smaller steps as the stochastic search grows closer to a maxima.

The stability results for the post-training controller are plotted in Figure 18. Note that stochastic action selection was used *during* training as well as for evaluation purposes. We see that the performance of the controller is almost identical to that obtained when a brute force search of the Q-function was performed, which implies that most of the time the simulated annealing approach successfully found the value-maximizing action. The number of steps before search termination is listed in Table I, and generally, several hundred steps were required before the annealing algorithm converged (this could possibly be improved by a more carefully chosen cooling schedule and/or step-generation technique). Convergence was declared when no improvement to the objective function had occurred for $N = 100$ steps.

3. Exploiting Temporal Coherence

One final caveat here is that simulated annealing, while well-suited to continuous action-select, is not known for its execution speed. In fact, the slower the algorithm proceeds, the more accurate the answer it generates [99]. We will here look at one possible method for speeding up this simulated annealing action selection that is especially suited to reinforcement learning / LSPI.

In many realms of real-time physical simulation, the concept of *temporal coherence* is used to speed up computation. In a simulation context, temporal coherence refers to the property of a system state in a given time-step to resemble the system state in recent, past time-steps. That is, very generally, if we were monitoring the state of a physical system as a function of time $\mathbf{x}(t)$, the central idea of temporal coherence could be expressed as $\mathbf{x}(t) \approx \mathbf{x}(t+1)$.

Since LSPI is often used to train controllers, we can exploit the idea of temporal coherence here to speed up our action search. That is, in many cases, we may suspect that temporally adjacent controller responses will be similar since the state of the system will be similar. As such, we may gain some performance advantage by beginning our stochastic action search at the point in action-space where the last optimal action was found.

4. Experiment 3: Simulated Annealing Action Search with Temporal Coherence

We performed Experiment 2, only each time our simulated annealing search converged on a state a_t, when simulated annealing was run for find the value-maximizing action for the next time-step $t+1$, the search was started at state $a_0 = a_t$. We measured stability and average number of steps before convergence as before, and the results

are plotted in Figure 18. Notice that the stability results are roughly the same as when the update rule from Experiment 2 was used (no speedup) and also roughly equivalent to the results when brute-force maximization was used. However, taking advantage of temporal coherence did offer a significant speedup in terms of number-of-steps required for algorithm convergence. We also performed this version of the experiment with the strong-wind forces turned off. In this case, we find the speedup even more significant, and this should not be too surprising, since the assumptions of temporal coherence are most accurate when the system is relatively 'predictable', and the system state is relatively unchanged from time-step to time-step, more or less the case when the system is only applying corrective balancing forces, but definitely *not* the case when the system must respond to strong *random* wind forces. The results of this experiment are listed in Table XI.

Table XI. Search steps with and without the consideration of temporal coherence

Domain	Avg. Steps
Wind, No Speedup	261 $\pm$ 7
Wind, With Speedup	175 $\pm$ 11
No Wind, No Speedup	260 $\pm$ 10
No Wind, With Speedup	150 $\pm$ 9

F. Conclusions

We have explored some of the difficulties encountered in using approximation-based reinforcement learning (specifically the Least-Squares Policy Iteration algorithm [23])

over continuous action spaces. We found that approaching the problem of action-selection as it is generally approached in the discrete-action domain can lead to significant algorithmic difficulties in narrow-Q domains (where all actions in a given state have very numerically very similar Q-values). We proposed a theoretically justified heuristic solution to the problem, and demonstrated the use of the heuristic to train a controller on a continuous-action domain.

In the spirit of the rest of this work, this approach is inherently an approximate one. By modifying the Q-values, one changes the learning environment and risks altering the quality of the final policy. However, in this chapter we derived bounds on these effects.

Our simulation domain consisted of a noisy inverted pendulum problem where the controller must be able to apply both small, gentle, corrective forces when the pendulum is near vertical equilibrium, and large corrective forces in response to large external noise forces. This domain was well suited to highlight the advantage of using continuous-action controllers, and we found that our continuous-action controller outperformed comparable discrete-action controllers. We then discussed the problem of practically searching for Q-maximizing actions in a continuous action space and proposed the use of simulated annealing to maximize the potentially very hilly, continuous-action Q function. After providing a framework for this search, we showed how the stochastic simulated annealing search could be sped up by exploiting temporal coherence in the response of our balancing controller.

CHAPTER V

ACCELERATED APPROXIMATE REINFORCEMENT LEARNING ON THE GPU

The final technique presented in this dissertation to speed up reinforcement learning (RL) in unconstrained multi-agent settings will be to "simply throw more computational power at the problem". In this chapter, we will specifically concentrate on speeding up the process of approximate policy iteration algorithms (and specifically the Least-Squares Policy Iteration algorithm (LSPI) [23]) that we have used extensively in previous chapters.

We will present a method for direct, hardware-based computational speedup for the two main parts of LSPI (policy evaluation and policy improvement) by parallelizing the algorithm for execution on the GPU. While this approach does not solve the scaling problems associated with using approximate reinforcement learning in a multi-agent context, there are numerous justifications for seeking this sort of direct computational speedup for multi-agent RL:

In the situations where we do not use any specially designed speed-up algorithms for multi-agent reinforcement learning (because such the speed-up algorithms are not applicable to the domain, for example), direct GPU-based parallelization will be possible.

And, when we *do* use specialized, tractable algorithms (such as the ones presented earlier in this dissertation), hardware-based acceleration will still be useful. Even in Coordinated Reinforcement Learning [59] or gradient ascent approaches like MALSPI (previous chapter), normal Least-Squares Policy Iteration is *still* performed in some context. That is, many efficient reinforcement learning algorithms effectively act as 'wrapper' algorithms around LSPI, and thus accelerating the core LSPI algorithm is

still broadly useful.

A. Previous Work / Background

1. Parallel Computation in Artificial Intelligence

Parallel processing techniques have seen mixed levels of adoption by the artificial intelligence community. On the one hand, certain aspects of artificial intelligence lend themselves very naturally to parallel implementations. For example, the execution of neural nets on parallel computers is a particularly obvious development, and was explored very early in the course of that vein of research [103]. Additionally, in multi-agent systems research, where one deals with inherently distinct autonomous agents which can be simulated on independent hardware, the benefits of parallel computation are fairly obvious and easy to exploit.

However, other aspects of artificial intelligence research such as many sub-disciplines of machine learning (the large exception being neural network research) have rarely seen parallel implementations. Specifically in the case of reinforcement learning, the literature is fairly void of research in this area. Some notable exceptions are Kretchmar's work [104] on multiple agents undergoing reinforcement learning in parallel (although that work did not specifically address parallelizing the reinforcement learning *algorithm* itself), and Likas' work on genetic/reinforcement-learning algorithms [105] (there a *novel* algorithm was proposed which was amenable to parallelization ... the research did not touch on methods to parallelize *known*, widely used and efficient reinforcement learning algorithms).

2. GPU Background

Graphical processing units have evolved from their origins as simple co-processors designed to speed up straightforward memory transfer operations (such as the ANTIC co-processor for the original Atari machines, which supported only simple BitBLT operations for sprites) to flexible, nearly stand-alone processors capable of performing complex parallel computations. Today's GPUs are SIMD (single instruction, multiple data) processors capable of executing programs (often referred to as shaders) at speeds on the order of tens of Gflops. Shaders come in several varieties such as pixel shaders (also known as fragment shaders) which operate on color texture data and vertex shaders, which operate on vertex data.

Although GPUs were initially designed for graphical purposes, and most early shaders were written for real-time, rendering-related tasks, modern hardware is general enough to support a wide variety of shader-based computation, and as such GPUs are being used increasingly for scientific and data processing purposes. Recent examples include applications such as finite-element simulations [106], modeling ice-crystal growth [107], cloud simulation [108], fluid-flow using the lattice Boltzmann model [109], etc. More mathematically oriented applications such as algorithms for matrix manipulation [110, 111, 112] have also been explored. An excellent overview of such such recent, non-graphical GPU-based algorithms can be found in [113].

B. Approach

In this chapter we will present an implementation of the popular Least-Squares Policy Iteration reinforcement learning algorithm [23] such that portions of the algorithm exploit parallelization and execute entirely on the GPU. Remember that at its core,

LSPI is a simple iterative algorithm with two main parts:

Step 1: A matrix **A** is constructed which takes into account the current policy π and the nature of the environmental state transition probabilitie. Essentially, this matrix captures *how* the agent interacts with its environment. A vector **b** which represents the reward function is also created in this step, but this vector only needs to be constructed once per run of the algorithm (not per iteration), and thus represents only a fixed computational cost we will not attempt to lessen here.

Step 2: The linear system $\mathbf{A}\omega = \mathbf{b}$ is solved for the weight vector ω. An improved policy π' can be constructed from this weight vector, which will be used to construct the next **A** in **Step 1** of the next iteration of the algorithm.

Since **A** is a $k \times k$ matrix (where k is the number of basis functions used in Q-value approximation), the computational effort in Step 2 is equivalent to the work required to invert a $k \times k$ matrix. For many problems k will be fairly small (especially if the basis functions are carefully chosen for the problem domain), and the effort required to invert **A** will be fairly manageable. Even in domains where many basis functions are required and **A** is large, **Step 2** is still a very straightforward inversion operation. Already a great deal of work has been done on efficiently solving linear systems of equations on the GPU including work on both Gauss-Jordan Elimination and LU-Decomposition (e.g. [114]), and we will certainly not contribute anything to this body of work in the context of this dissertation.

As such, as we will concentrate specifically on speeding up the first step: the construction of the matrix **A**. We will not consider the construction of **b** further since its construction represents only a constant fixed cost to the policy iteration

process.

Roughly, the effort involved in the construction of **A** scales with the number of basis functions k, but also with the number of training samples (experience tuples) used to train the agent. For even simple problems, the number of training points can be large (several hundred thousand data points) [59, 23], and for multi-agent domains, this number can grow well into the millions [26]. Since higher numbers of training data points result in higher transition probability approximation accuracies [23] and thus potentially in a more optimal post-training policy, the computational demands of **Step 1** should be viewed as scaling with the *quality of the desired solution.*

1. Factoring A

The expression for **A** is fairly complex, but we can simplify the expression if we introduce the two matrices **X** and **Y**:

$$\mathbf{X} = \left[\vec{\phi}(s_1, a_1), \ldots, \vec{\phi}(s_N, a_N)\right] \tag{5.1}$$

$$\mathbf{Y} = \left[\vec{\phi}(s'_1, a'_1 = \pi(s'_1)), \ldots, \vec{\phi}(s'_N, a'_N = \pi(s'_N))\right] \tag{5.2}$$

such that the matrix **A** can be defined as:

$$\mathbf{A} = \mathbf{X}(\mathbf{X} - \gamma\mathbf{Y})^T \tag{5.3}$$

This particular factorization will come in useful later, as the two matrices **X** and **Y** will be easier for us to compute separately.

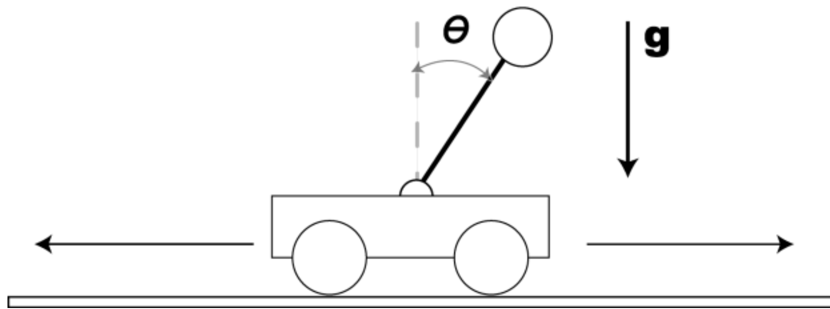

Fig. 19. Force diagrams for the cart-pole balancing problem.

C. Basis Functions in LSPI

While the basis functions used in LSPI can be anything (step functions, spherical harmonic functions, etc.), generally the functions are chosen in such a way as to ensure smooth and consistent coverage of the state/action space. This is generally necessary since there is usually little *a priori* knowledge about the structure of the state/action space (after all, reinforcement learning is intended for use in situations where human designers have difficulty hand coding an appropriate policy for the domain).

For example, in the original paper where LSPI was presented [23], the authors tackled the problem of using LSPI to train a controller for the pole balancing problem. This problem has two variables, $x_1 = \theta$, which is the angular deflection of the pole from the upright position, and $x_2 = \dot{\theta}$, which is the angular speed of the pole. The agent has access to three actions - push left (a_1), push right (a_2), and don't push (a_3), and the goal is to keep the pole balanced upright for as long as possible (these quantities are illustrated in Figure 19). The authors used 27 radial basis functions spaced evenly over the state/action space after the form (illustrated in Figure 20):

$$\phi_i(s, a) = \phi_i(x_1, x_2, a) \in \{e^{-((x_1-c_1)^2+(x_2-c_2)^2)/3}\delta_{a,\, c_3}\} \tag{5.4}$$

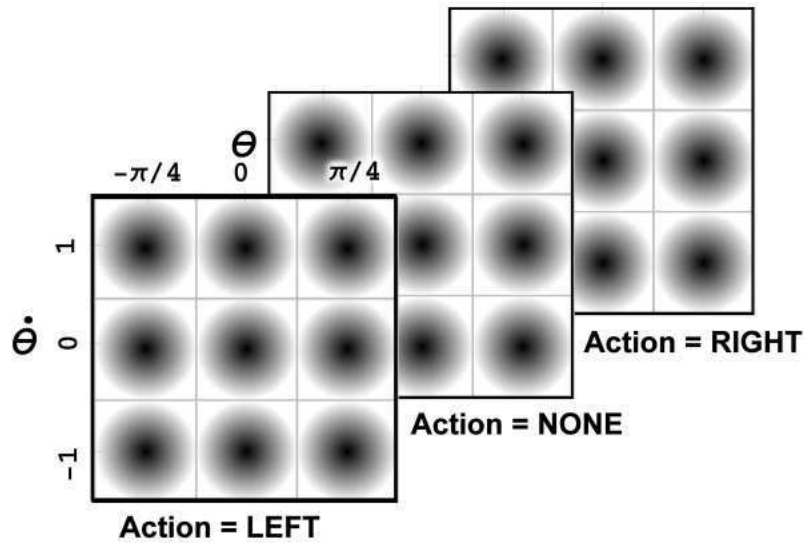

Fig. 20. Illustration of the basis functions used in the cart-pole balancing example.

where $c_1 \in \{-1, 0, 1\}$, $c_2 \in \{-1, 0, 1\}$, $c_3 \in \{a_1, a_2, a_3\}$. This series of basis functions was chosen because the authors knew that x_1 and x_2 would *generally* lie in the range $x_1, x_2 \in [-1, 1]$, and so spacing basis functions over this area seemed like a good idea, and in fact resulted in a very successful post-training policy [23].

It is important to note here that the authors chose to use a delta function for the action-dependent part of the basis function. This is fairly common in LSPI problems, and with good reason ... if delta functions are used for the action-dependencies, maximization of the Q-function (again, which is necessary to determine which action is most desirable to take next) reduces to the following operation: Imagine that our basis functions are of the form $\phi(s, a) = \psi(s)\chi(a)$ and that they are indexed by two parameters i and j, one for the 'state' functions $\psi_i(s)$ and the other for the 'action' functions $\chi_j(a) = \delta_{a,j}$ (we will assume that there are m state-functions $\psi_i(s)$ and n action-functions $\chi_j(a)$):

$$\phi_{i,j}(s,a) = \psi_i(s)\delta_{a,j} \tag{5.5}$$

Then we have:

$$\arg\max_a \hat{Q}(s,a) = \arg\max_a \sum_{i,j} \psi_i(s)\chi_j(a)\omega_{ij} = \arg\max_a \sum_{i,j} \psi_i(s)\delta_{a,j}\omega_{ij} \tag{5.6}$$

$$\arg\max_a \hat{Q}(s,a) = \arg\max_a \sum_i \psi_i(s)\omega_{ia} \tag{5.7}$$

and if we let:

$$Q_a(s) = \sum_i \psi_i(s)\omega_{ia} \tag{5.8}$$

then maximizing $\hat{Q}$ reduces to:

$$\arg\max_a \hat{Q}(s,a) = \arg\max Q_a(s) \tag{5.9}$$

which is a simple operation involving n comparisons, where n is the number of actions.

The nice thing here is that no matter how many actions we have, we only need to compute the state-basis functions $\psi_i(s)$ *once* (these could be quite complicated, after all) ... then, to get the individual $\hat{Q}(s,a)$ we only have to multiply these values with with portions of the weight vector ω.

Because of this simplifying property, it is fairly common in the literature to structure the basis functions in this way (for example, *all* the examples presented in [23] are structured this way), and as such, we will here work from the assumption

that our basis functions assume the form:

$$\phi_{ij}(s, a) = \psi_i(s)\delta_{a,j} \tag{5.10}$$

D. Brook Architecture

Brook is a high-level language for developing GPU-accelerated applications [115]. It allows programmers to write software in a familiar C-style environment, and takes care of translating that high-level code into low-level fragment shader instruction form. Specifically, Brook is a stream-based language, which means that much of Brook is designed around performing repetitive calculations on large amounts of similarly structured data, which makes sense for a language designed to develop programs for SIMD-like GPUs.

1. Streams

All operations in Brook are performed on *streams* of data. Streams are roughly analogous to arrays in C, except that operations cannot be performed on Streams in a random access fashion. Instead, operations are atomically applied to all elements of a stream. Practically, when an operation is applied to a stream, it is executed as a render pass on the GPU, and a computational speedup is (hopefully) obtained because of the GPU's ability to operate on multiple stream elements simultaneously.

2. Kernels and Reductions

Brook has two main categories of code that can be run on streams: kernels and reductions. Kernels take multiple streams as input and produce a single output stream. The dimensionality of the input streams and output streams must be identical ... if the dimensions of the input/output streams differ, the streams are replicated until dimensionality is uniform across all input and output streams.

For example, if we were performing a kernel operation on a $N \times 1$ column vector (stream) $\mathbf{A}$ and a $1 \times M$ row vector (stream) $\mathbf{B}$, we could produce an $N \times M$ output matrix $\mathbf{C}$ with the following kernel (which implements the outer product $\mathbf{C} = \mathbf{AB}^T$):

```
kernel outer_product(input stream A, input stream B, output stream
C) {
    C = A * B
}
```

the kernel would run for each output element of $\mathbf{C}$, and the input streams $\mathbf{A}$ and $\mathbf{B}$ would be accessed appropriately.

Reductions, on the other hand, are used to reduce the dimensionality of input streams and produce a lower-dimensional output stream. As such, reductions can be used for operations such as summations, multiplying all the elements in a matrix, searching for the maximum element in a stream, etc.

Reductions can only accept one input argument and only produce a single output stream. In general, the input stream and output stream are different sizes, and groups of input elements (input cells) are mapped to single elements of the output stream. For example, we could perform a column-summation operation on a square, $N \times N$ matrix $\mathbf{A}$ (and generate, say, a $1 \times N$ row vector $\mathbf{B}$ of column sums) by performing the following reduction:

```
reduction summation(input stream A, output stream B) {
    B += A
}
```

Which dimensions get reduced is determined by the size of the input/output streams, which are defined before the reduction is executed.

E. Description of the Algorithm

We now describe our process for speeding up the construction of the **A** matrix:

1. Step 1: Loading the Experience Tuples

Step 1: Loading the Training Corpus

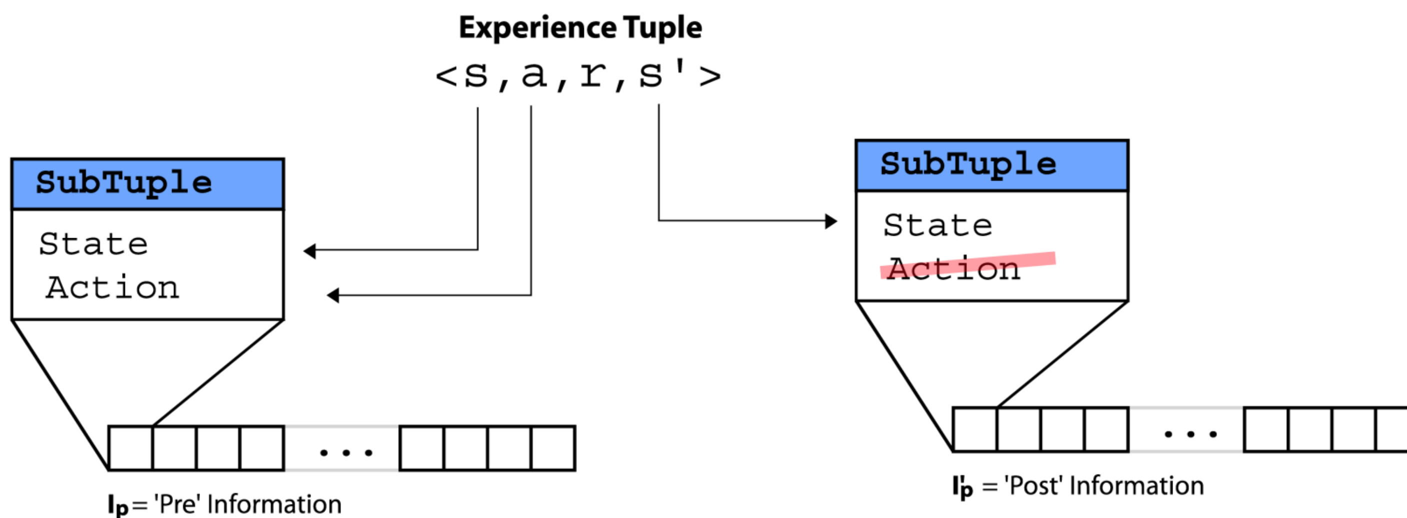

Fig. 21. Step 1 of the LSPI algorithm in Brook. Data from the training corpus is converted into two input streams. Note that the 'post' input stream has no action information. This is because this action information must be *computed* ... specifically, this information represents the actions that would be taken in states s' under the current policy π.

Illustrated in Figure 21, the first step in computing the matrix **A** is to load the N experience tuples into the video memory of the GPU for further processing. In a Brook context this simply means that we have to create streams which contain this training information.

This process can potentially be time-consuming, since the number of experience tuples N can be very large and data transfer from the CPU to the GPU across the graphics bus is generally relatively slow. However, like the construction of the **b** matrix, this process needs only occur once per run of the entire policy iteration algorithm - not on a per iteration basis - and thus represents only a fixed cost.

Again, because we'll be implementing the algorithm in Brook, we need to express the experience tuples as stream data. The most straightforward way to do this is to create a stream of structures containing the tuple information. Since we're only going to be constructing the matrix **A** and not the reward-influenced **b** vector, we will only need the state and action information from the experience tuples. For reasons we will see in a moment, we will split the experience tuple data into what we will call 'pre'-reward information I_p and 'post'-reward information I'_p. The 'pre' information I_p will be defined as:

$$I_p = \langle s, a \rangle \tag{5.11}$$

or roughly, as what state the agent *was* in and what action the agent *took*. The 'post' information will be defined as;

$$I'_p = \langle s', \pi(s') \rangle \tag{5.12}$$

or, roughly, as what state the agent *ended up in* and what action the agent *will* take if it follows its current policy π. A complete (over-complete actually) experience tuple

can be re-created by combing pre, post, and reward information as:

$$T = \langle s, a, r, s' \rangle \in \langle I_p, r, I'_p \rangle \tag{5.13}$$

The benefit of separating an experience tuple in this way is that the pre and post information will need to be processed separately, and in addition, both pre and post information share a common composition. That is, both pre and post information can be represented by a state part and an action part:

$$I_p, I'_p = \{STATE, ACTION\} \tag{5.14}$$

If we continue the example of the pole-balancing problem that we discussed earlier, we might create the following pre/post information structure in Brook:

```
struct SubTuple {
    float x1;
    float x2;
    float action;
}
```

Note that we have used floats to represents even the discrete variable 'action' - this is because in Brook all streams must be defined entirely in terms of floats. As such, we can represent the training corpus as two $1 \times N$ streams of SubTuple information:

```
SubTuple pre_information<1, NUM_SAMPLES>;

SubTuple post_information<1, NUM_SAMPLES>;
```

Also note that at this point, the 'post' information stream I'_p contains no action information. This is because the action information in the 'post' information stream

(the action that the agent *would* take in the state s' under the current policy) is simply not included in the training data since this information is a function of the current policy.

2. Step 2: Start Calculating the 'Post' Actions

Illustrated in Figure 22, the eventual goal at this point is to fill in the missing action information from the 'post' stream. As a necessary intermediate step to that goal, we will create vectors of basis functions from the 'post' sub-tuple information stream. As it will turn out, for each element of the post information stream, we need to generate the vector $\vec{\Phi}(s)$:

$$\vec{\Phi}(s) = \begin{bmatrix} \phi_{1,1}(s, a_1) \\ \phi_{2,1}(s, a_1) \\ \vdots \\ \phi_{m,n}(s, a_n) \end{bmatrix} \tag{5.15}$$

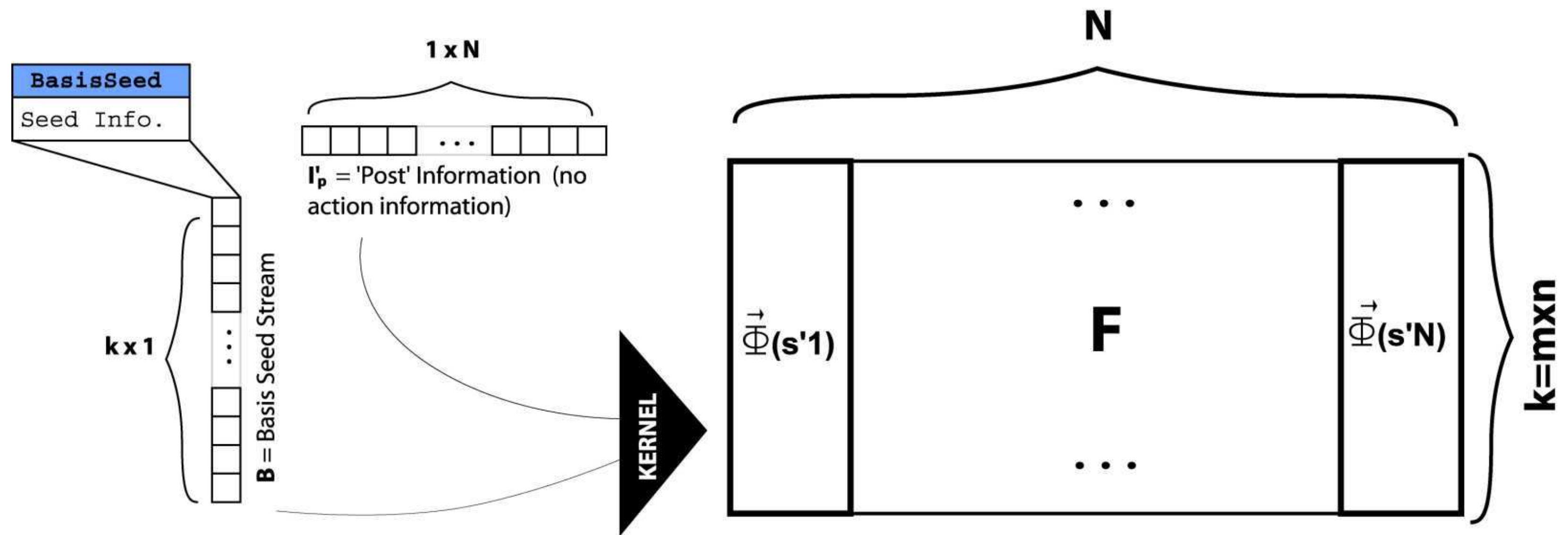

Fig. 22. Step 2 of the LSPI algorithm in Brook. The state/action information from the 'post' information stream $\mathbf{I}'_{\mathbf{p}}$ is used in conjunction with the basis seed information to create a matrix of basis functions $\mathbf{F}$.

Currently our 'post' information stream is of shape $1 \times N$. After processing - after creating a $\vec{\Phi}$ vector from each element of the stream, this will become a $(k = m \times n) \times N$ stream (N, k-element $\vec{\Phi}$ vectors tiled side by side) which we will call $\mathbf{F}$. In order to do this in Brook, we will run a kernel over the 'outer product' of the row-shaped, N element 'post' information stream I'_p and a column-shaped, k-element stream $\mathbf{B}$ containing information about the basis functions. The stream $\mathbf{B}$ is necessary since a single kernel procedure must be able to compute *all* the basis functions ... this $\mathbf{B}$ stream provides 'seed' information to allow a single kernel program to simulate all the basis functions.

To make this more clear, let's continue the pole balancing problem presented earlier ... in the original presentation paper [23], the authors constructed basis functions of the form:

$$\phi_i(s,a) = \phi_i(x_1, x_2, a) \in \{e^{-((x_1-c_1)^2+(x_2-c_2)^2)/3}\delta_{a,\, c_3}\} \tag{5.16}$$

such that each basis function had three parameters c_1, c_2 and c_3. As such, a unique instantiation of these three c values yielded a unique basis function. In order to allow a single kernel procedure to compute this entire family of basis functions, we need a k-element column vector $\mathbf{B}$ such that each element has enough information to allow us to instantiate all the parameters of a basis function, and will use the term basis *seed* to represent this information. In the pole balancing case, a basis seed information structure might be something like:

```
struct BasisSeed {
    float c1;
    float c2;
    float c3;
```

```
}
```

with the corresponding **B** column stream (27x1):

$$\mathbf{B} = \begin{bmatrix} \{-1,-1,\text{LEFT}\} \\ \{-1,0,\text{LEFT}\} \\ \{-1,1,\text{LEFT}\} \\ \{0,-1,\text{LEFT}\} \\ \vdots \\ \{1,1,\text{RIGHT}\} \end{bmatrix} \tag{5.17}$$

3. Step 3: Action Determination

Illustrated in Figure 3, before we can fully process the 'post' information stream, we need to know the post actions $a' = \pi(s')$. That is, for each element in the post information stream, we have s' from the training corpus, but we will also need to compute what action would be taken under the current policy π in each state s'.

Because, as discussed earlier, the policy always chooses the action which maximizes the Q-function, we can figure out these $a' = \pi(s')$ actions by computing the method we discussed earlier ... since we have factored our basis functions as we have, we need only compute the $Q_a(s')$ values and then simply choose the action corresponding to the largest $Q_a(s')$.

In order to compute the $Q_a(s')$, we first performed **Step 2**, creating a basis-function matrix **F**. In order to obtain Q-values from the $\phi_{i,j}(s,a)$ values in **F**, we need to multiply each $\phi_{i,j}$ element by its corresponding $\omega_{i,j}$ weight value. This can be accomplished by a simple multiplication kernel. Specifically, we will perform a per-element multiply of the $k \times 1$-element weight vector ω and the $k \times N$ basis-function matrix **F** to yield a $k \times N$ matrix of Q-values, which we will simply denote **Q**. In

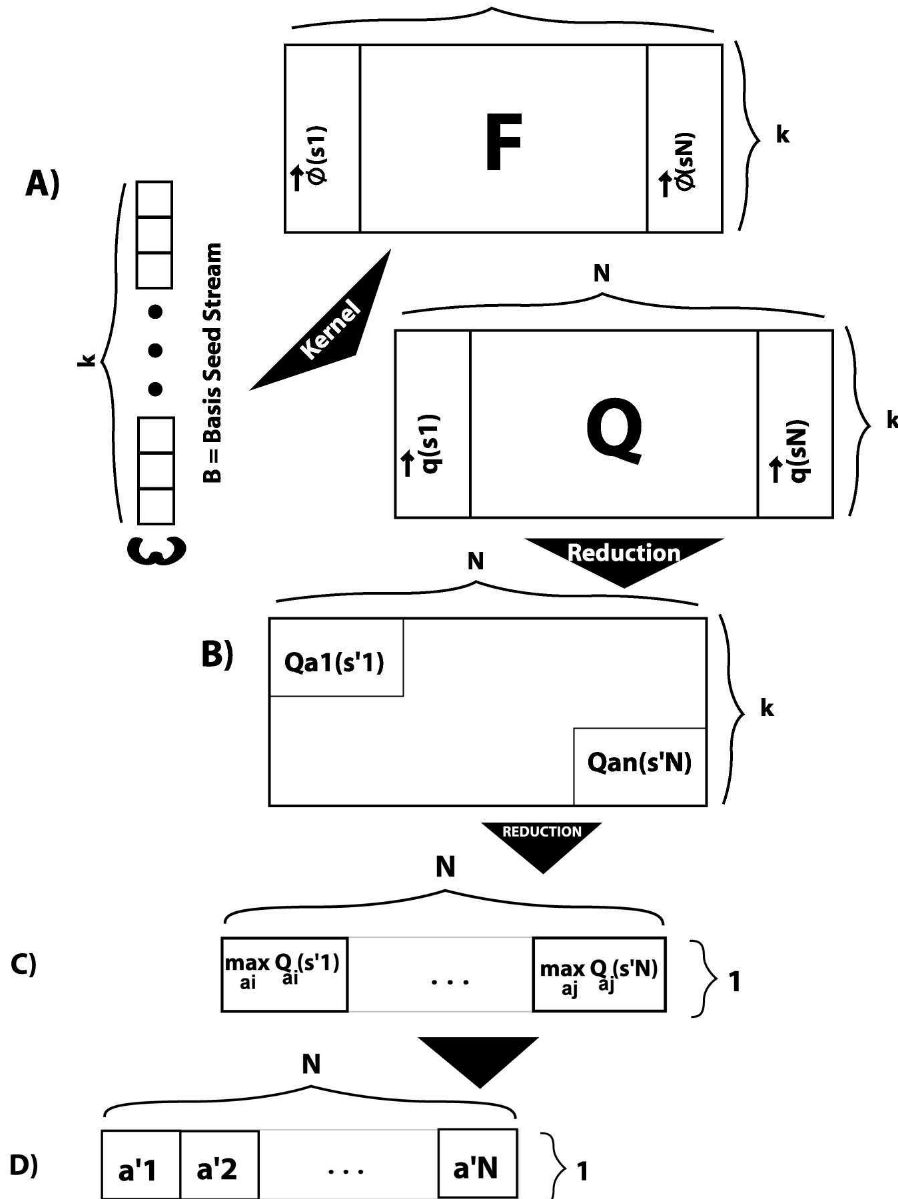

Fig. 23. Step 3 of the LSPI algorithm in Brook. A) The basis vector stream **F** is per-element multiplied with the weight vector ω to produce Q-values. B) These Q-values are cell-summed to yield the $Q_a(s')$ discussed earlier. C) These $Q_a(s')$ are compared to find the maximum. D) The index of the maximum $Q_a(s')$ is recorded ... this 'action index' is $\pi(s')$.

particular, after multiplication, we transform every $\vec{\Phi}(s)$ into a 'weighted' version $\vec{q}(s)$:

$$\vec{q}(s) = \begin{bmatrix} \phi_{1,1}(s, a_1)\omega_{1,1} \\ \phi_{2,1}(s, a_1)\omega_{2,1} \\ \vdots \\ \phi_{m,n}(s, a_n)\omega_{m,n} \end{bmatrix} \tag{5.18}$$

To evaluate the quantities $Q_a(s)$, we can then perform a 'cell-based' summation operation, summing over rows $1 \to m$ for $Q_1(s)$, $m + 1 \to 2 * m$ for $Q_2(s)$, etc., since by the definition of $Q_a(s)$ earlier, we have:

$$Q_{a_j}(s) = \sum_{i=(j*m)+1}^{(j+1)*m} \vec{q}(s)_i \tag{5.19}$$

Where we have made use of the fact that for any given a_j, many of the $\phi_{i,j}(s, a)$ are zero (because of the delta function). This takes us from a $(k = m \times n) \times N$ matrix $\mathbf{Q}$ to a $n \times N$ stream of $Q_a(s')$ values (**Step 3b**). We then need to find the maximum $Q_a(s')$ in each column (**Step 3c**), by a reduction operation, and then determine the row index of the element where each maximum $Q_a(s')$ appears (**Step 3d**), which is performed by a kernel operation followed by a reduction. After this we are left with a $1 \times N$ row-shaped stream of action $a' = \pi(s')$ values.

4. Step 4: Computing Pre and Post Basis Functions

Illustrated in Figure 24, now that we have complete 'pre' and 'post' information streams, we can go about computing the 'pre' and 'post' basis functions.

Similar to what we did before, we can run a kernel 'outer product'-fashion over

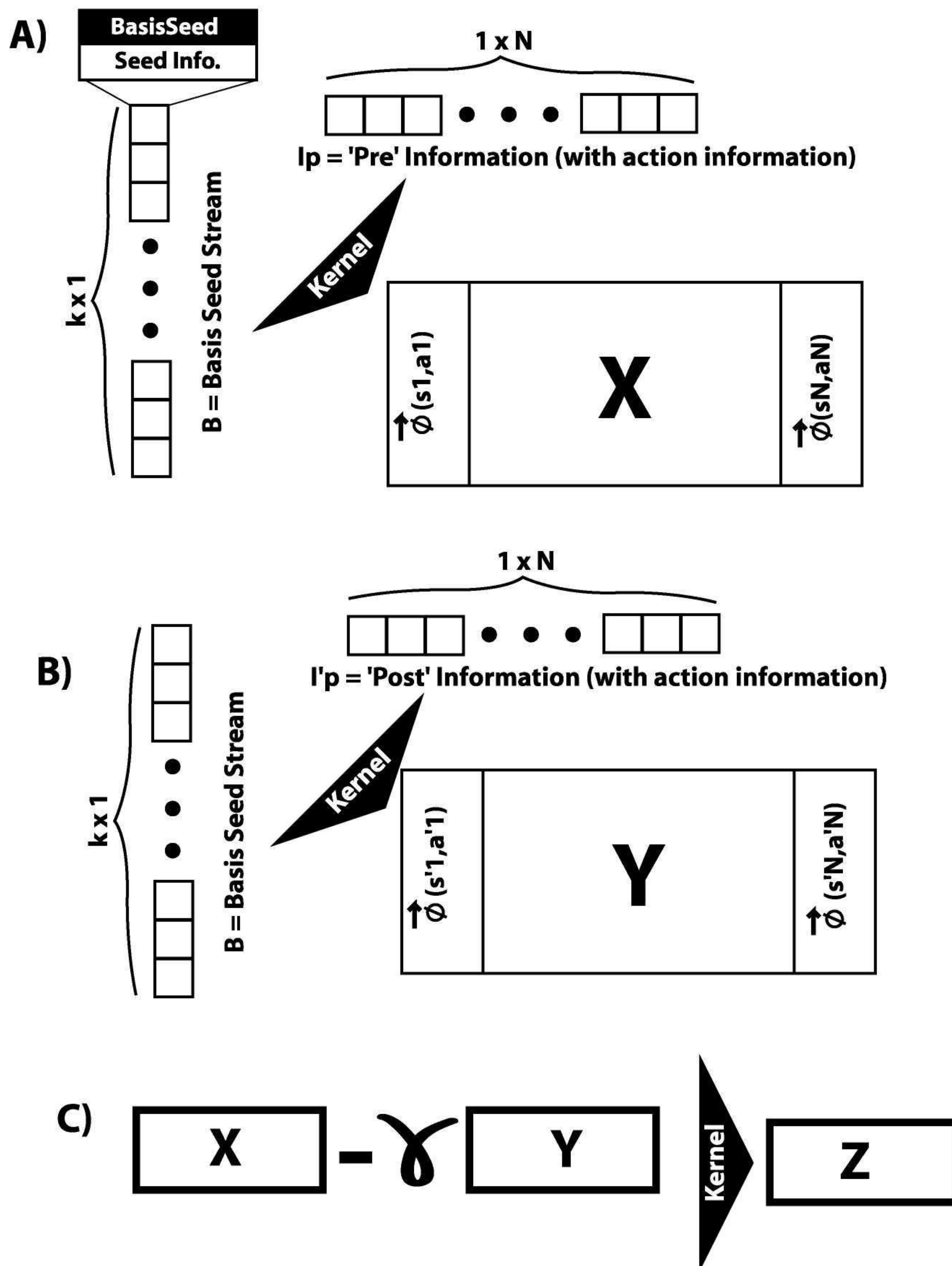

Fig. 24. Step 4 of the LSPI algorithm in Brook. In this step we calculate basis vectors for both the 'pre' and 'post' information streams.

the basis seed column stream **B** and the pre and post information row streams, only this time we compute regular basis vectors $\vec{\phi}(s,a)$ for each stream element (as opposed to computing $\vec{\Phi}(s)$ earlier).

$$\vec{\phi}(s,a) = \begin{bmatrix} \phi_{1,1}(s,a) \\ \phi_{2,1}(s,a) \\ \vdots \\ \phi_{m,n}(s,a) \end{bmatrix} \tag{5.20}$$

We generate a basis vector for each element of the 'pre' information stream, resulting in a $k \times N$ matrix **X**, and a basis vector for each element of the 'post' information stream, resulting in a $k \times N$ matrix **Y**. At this point we have the two matrices **X** and **Y** needed for the construction of **A** as discussed earlier. To speed up computation however, we will also compute an intermediate stream:

$$\mathbf{Z} = \mathbf{X} - \gamma \mathbf{Y} \tag{5.21}$$

such that

$$\mathbf{A} = \mathbf{X}\mathbf{Z}^T \tag{5.22}$$

5. Step 5: Computing A

Illustrated in Figure 25, finally we can go about computing the final matrix **A**. A single column of **A** can be computed by multiplying every row of stream **X** with a given row from stream **Z** (the ith row of **Z** if one wants to compute the ith column of **A**). This results in a $(k \times N)$-sized stream, which then needs to be summed along its

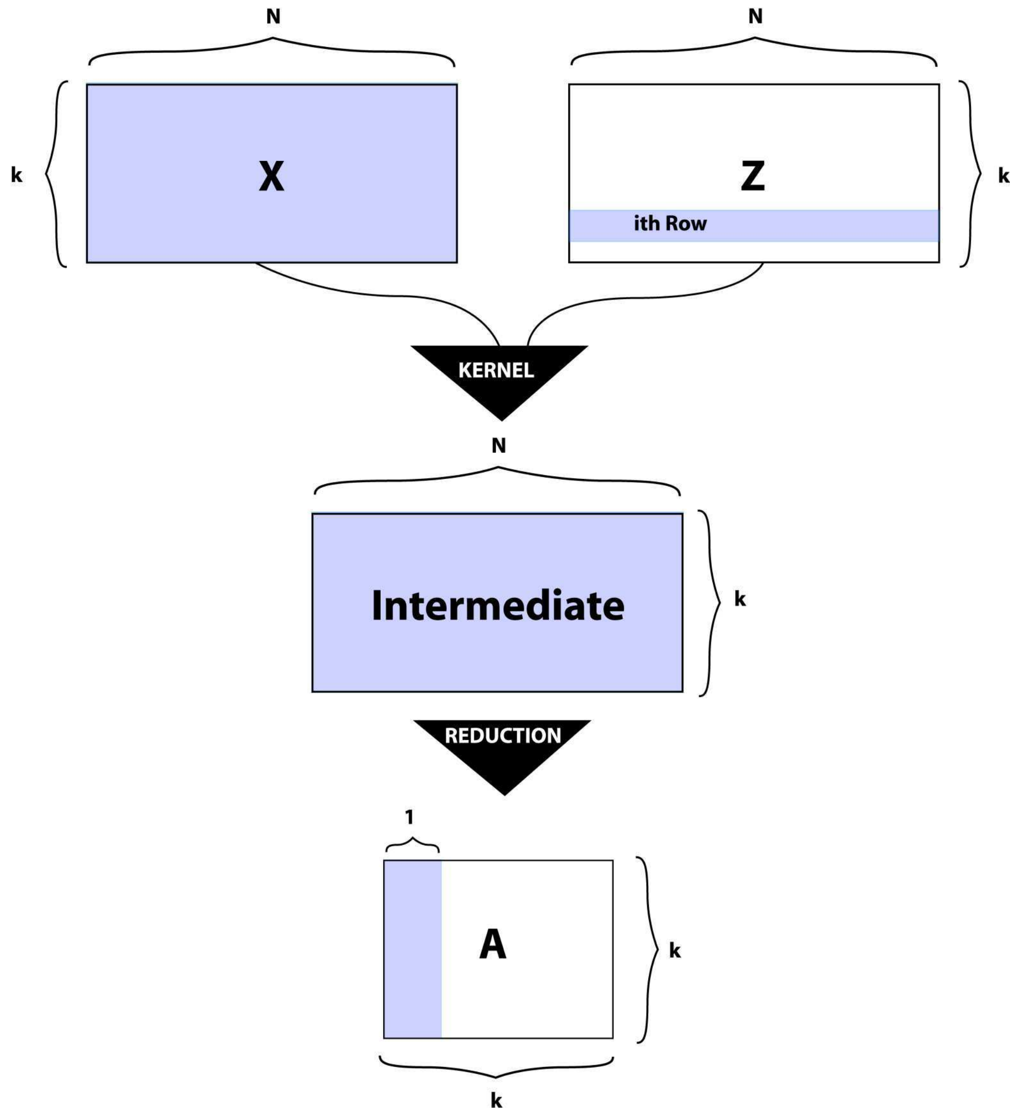

Fig. 25. Step 5 of the LSPI algorithm in Brook. Here we need to take the outer product $\mathbf{XZ}^T$. The most straightforward way to do this is to run a multiplication kernel over $\mathbf{X}$ and row i of $\mathbf{Z}$, which will generate a stream which can be column-wise summed to yield column i of $\mathbf{A}$. In practice, several columns of $\mathbf{A}$ are computed simultaneously (batched in the intermediate stream then summed at the same time) for efficiency.

columns to obtain an $(k \times 1)$, column-shaped stream which becomes the ith column of the matrix $\mathbf{A}$.

This summation operation sums over the large N-element dimension, and because the addition is performed in parallel, requires $O(\log_2 N)$ passes. While this will be a fairly small number, we would still like to minimize the total number of passes in the algorithm, and so to speed up computation, b number of columns of $\mathbf{A}$ are 'qued-up' in an intermediate stream and then the summation process calculates all b columns of $\mathbf{A}$ at once. As such, the intermediate stream is actually a $bk \times N$-sized stream (see Figure 7).

F. Experiment

To compare the performance of our Brook-based $\mathbf{A}$-generation scheme, we compared its performance against MATLAB and pure $C++$ versions of the same code. The test domain in all cases was the pole balancing problem [23] used frequently in the rest of this dissertation.

Specifically, as before, the comparison task was strictly the *construction* of the matrix $\mathbf{A}$. That is, actually *using* the $\mathbf{A}$ matrix in a real LSPI training run would involve first making the experience tuples available to whatever processor was building $\mathbf{A}$ (in the case of GPU-based construction, for example, this would involve data transfer across the graphics bus), then inverting the matrix post-construction, etc. Here we concentrate only on the *construction* of $\mathbf{A}$ and assume that, in the case of GPU-based construction, we already have access to the experience tuples and don't need to worry about inversion. This is a reasonable assumption since again, the experience tuples would only have to be loaded once per run of the policy iteration

algorithm, and thus represent only a constant, fixed cost to the process. Also, we do not consider transferring the matrix **A** from video memory back to system memory for further processing, as would be necessary if subsequent matrix inversion was performed on the CPU. This is also reasonable because the matrix **A** is relatively small (only $k \times k$ - perhaps on the order of a few million floats at most), and additionally, the matrix **A** doesn't *have* to be transferred at all since the linear system *can* be solved on the GPU anyway [114, 110].

We examined the performance of all three methods for constructing **A** with various numbers of experience tuples, and for various numbers of basis functions. Specifically (so that we could adequately test the performance of the algorithm as a function of number of basis functions) we implemented the multi-agent version of the pole balancing problem (identical to the case in Chapter III), where we allow multiple agents to apply force simultaneously to try and balance the pole. We implemented no special dimensionality reduction techniques (such as employing MALSPI), and so the state space, and thus the number of basis functions, scaled exponentially with the number of agents in the training domain. We simulated the case of 1,2,3, and 4 balancing agents, which meant respectively 20,40,80, and 160 basis functions were used.

For the MATLAB-based simulation, we used the same code used in Chapter III to train the multi-agent pole-balancing agents. For the C++ implementation, the MATLAB commands were directly translated into C++, however, in all cases, the hand-written C++ code performed worse than the MATLAB version, and as such, we omit the results for the C++ simulations. The MATLAB (and C++) code was executed on a 3GHz Pentium IV machine.

The Brook code used was exactly as listed in the subsequent 'Code Listing' section, with appropriate modifications made for increasing numbers of agents and

basis functions. The Brook code was executed on a 2.4GHZ Pentium IV machine with a nVIDIA GeForce6 6800 video card with 128 MBs of video RAM.

1. Results

The first thing we looked at is how the MATLAB and the GPU implementations scaled computationally with the number of experience tuples used to construct $\mathbf{A}$. Figure 8 shows how computation time varies with number of samples used in construction. Note that while the CPU implementation approaches a limit of no computation time as the number of training samples goes to zero, the GPU implementation approaches a constant fixed cost as the number of training samples decreases. This is most likely due to the fixed costs associated with setting up and executing the various render passes used in the algorithm (even if they operate on no data). Even though the GPU implementation has this fixed cost, it scales much more slowly with increasing numbers of training samples than does the CPU version, and the 'break-even' point is at about 400 training samples (Figure 26).

Note that the plot only extends to $N = 2000$ training samples. This is because the graphics card we ran the simulation on had a maximum texture size of 2048×2048 pixels, limiting our pre and post information streams to 2048 elements. This does not mean that the GPU-based algorithm could never be run on more than 2048 experience tuples, only that a given *run* of the algorithm could only process 2048 elements at a time. Because of the way $\mathbf{A}$ is constructed, it is be trivial to process 'batches' of training samples, and then combine the individual $\mathbf{A}$s produced by each batch: in fact, if each 'batch' produced a matrix $\mathbf{A_i}$, the $\mathbf{A}$ for all the batches would simply be $\mathbf{A} = \mathbf{A_1} + \mathbf{A_2} + \ldots$ [23].

Next we explored how the CPU and GPU implementations of the LSPI algo-

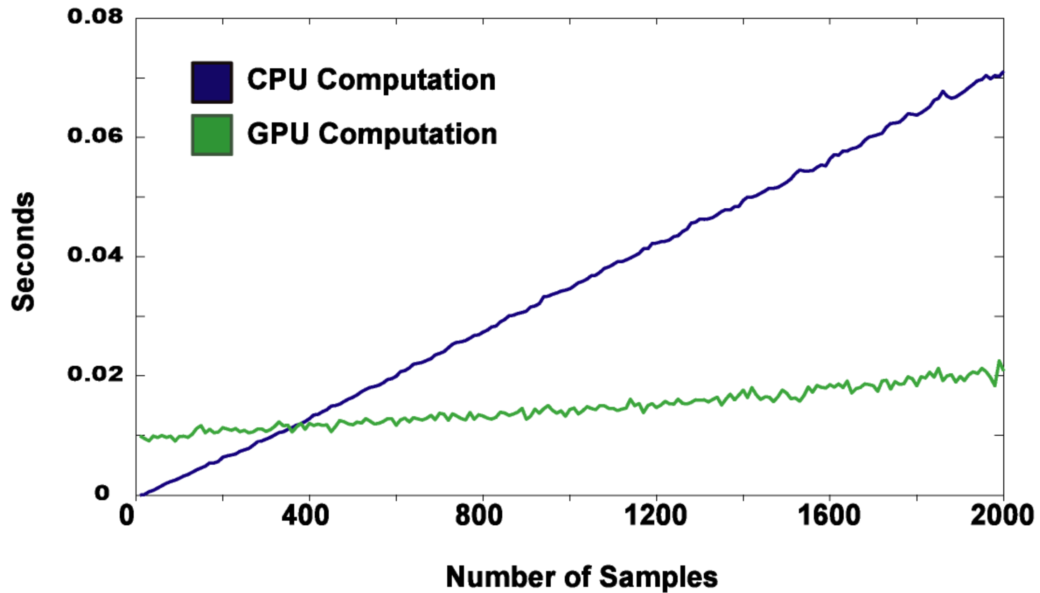

Fig. 26. Benchmark of the Brook implementation of the LSPI algorithm. Here we plot how the computational requirements of the CPU and GPU-based versions of the LSPI algorithm vary with the number of training samples used. Note that the plot only extends to 2000 training points. This is because the maximum texture size on our test graphics card was 2048 pixels. This doesn't imply that the GPU-based algorithm could never process more than 2048 training samples, only that no more than $N = 2048$ training samples could be processed in any given run.

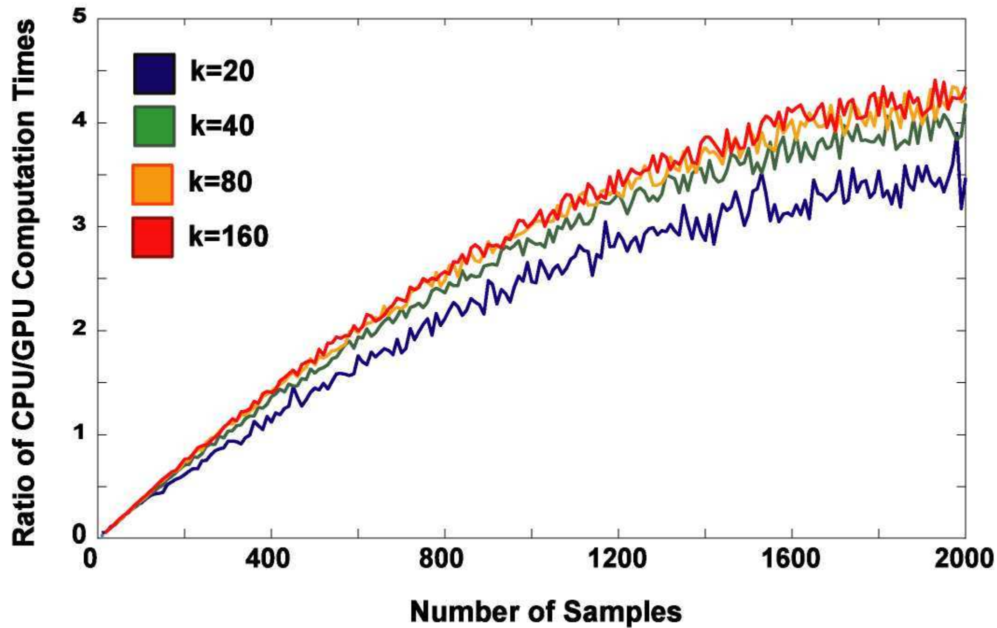

Fig. 27. Second benchmark of the Brook implementation of the LSPI algorithm. Here we plot the ratio T_{cpu}/T_{gpu}, where T_{cpu} is the time required to construct $\mathbf{A}$ on the CPU (MATLAB) and T_{gpu} is the time required to construct $\mathbf{A}$ on the GPU. We also vary the number of basis functions used. Better relative performance is obtained when more basis functions are used, though the ratio seems to asymptotically approach something like 4.

rithm performed as a function of the number of basis functions used (Figure 27). As mentioned before, we ran the algorithm with $k = 20, 40, 80$, and 160 basis functions and compared the time required to construct $\mathbf{A}$ - we plot the results in Figure 9.

Specifically, in Figure 2 we plot the T_{cpu}/T_{gpu}, where T_{cpu} is the time required to construct $\mathbf{A}$ on the CPU and T_{gpu} is the time required to construct $\mathbf{A}$ on the GPU. The four curves represent the four different numbers of basis functions we used, and better relative performance was seen when higher numbers of basis functions were used.

Once again, the plot only extends to $N = 2000$, though by the curvature of the graph, we expect not much more relative performance gain would have been

extracted by using a higher N anyway. Notice also that the limit of the curves seems to be asymptotically approaching something like 4.5 with increasing number of basis functions.

2. Discussion

The fact that we saw algorithmic performance increase with increasing numbers of training samples and basis functions is not surprising since the implementation proposed in this chapter parallelizes the construction of $\mathbf{A}$ over both basis functions and training examples. Further, simple calculations show that this simple Brook-based implementation came surprisingly close to the theoretical maximum GPU/CPU speed ratio.

According to NVidia-written documentation, the GeForce 6800 Ultra (the GPU used in our experiments) can perform at a maximum of 40 Gflops, while a 3Ghz Pentium 4 can perform at only at 6 Gflops [116]. This would mean that our GPU-based algorithm could have *only* run 5.8 times faster than a pure CPU based algorithm. Our experiments show that we approach something like a 4.5 CPU/GPU execution time ratio.

G. Conclusion

Unlike the techniques presented in previous chapters, this LSPI implementation is not an approximation-based method. That is, there is no additional error introduced into the RL algorithm via this technique - it is entirely a hardware-based speedup. As such it deviates slightly from the spirit of the rest of this dissertation. Nevertheless, it is,

strictly speaking, a method that would be quite useful when applying reinforcement learning in unconstrained multi-agent domains. As we discuss in the introduction, even though this technique does not change the computational complexity of the RL process, it does offer a speedup, and there are many likely situations where this speedup can be applied even in conjunction with specialized algorithms which *do* change the computational complexity of the multi-agent RL task.

As such, we introduced a GPU-based implementation of the standard LSPI algorithm and demonstrated its use on a standard pole-balancing RL task. The implementation parallelizes the construction of the **A** matrix used in the LSPI algorithm, and parallelizes this construction over both basis functions and training examples. We saw, as expected, that relative performance (over a pure CPU-based implementation) increased as a function of both basis functions and training samples used. We finally noted that our algorithm came achieved a CPU/GPU execution-time ratio of 4.5, which is close to the optimal ratio of 5.8.

Code for the Brook implementation and the basis seeds are listed in Appendix A.

CHAPTER VI

CONCLUSION

Throughout this dissertation, we have explored ways to reduce the computational complexity of reinforcement learning in multi-agent environments. Previous work, such as Guestrin's Coordinated Reinforcement Learning [59], and Wolpert's reward modification method [53] has been done to address this issue, but in this dissertation we concentrated on how to reduce the complexity of RL in the 'unconstrained' mutli-agent domain, that is, when we don't know much about our agent learning system before training commences.

We first demonstrated that a large reduction in computational complexity can be achieved by first segmenting an agent population under training into non-cooperative 'coalitions' and training each coalition separately. The problem, in unconstrained domains is that it is not obvious, pre-training, which coalitions should be formed. We introduced an algorithm called Information-Based Coalition Formation (IBCF), which used information-theoretic methods to form coalitions pre-training directly from training data. The algorithm worked by forming coalitions such that the agents in each coalition possessed as much information as possible about how states and actions related to rewards. We also proved optimality theorems for our algorithm. Specifically, we showed how one can ensure optimally performing coalitions post-training by ensuring that pre-training each coalition has a maximum amount of information about how states and actions relate to reward. IBCF is a greedy algorithm, and as such cannot guarantee that each coalition's information is actually maximized. As such, we proved error bounds for the case when a coalition's information is not actually maximized but comes close to being maximized. Finally, we demonstrated IBCF on several multi-agent training domains. In all cases the algorithm performed compa-

rably to peer-algorithms such as Coordinated Reinforcement Learning, except that IBCF required no pre-knowledge about the agent training system.

We next addressed the problem of the sample complexity of multi-agent reinforcement learning. The sample complexity of a learning problem is a measure of how many learning examples are required by a learner to, with a high probability, learn a given concept. After discussing how the sample complexity of multi-agent reinforcement learning grows exponentially with the number of system agents, we introduced a mechanism to perform 'hybrid' approximate reinforcement learning. 'Hybrid' learning is a general concept in machine learning where one offsets a lack of training examples by integrating domain knowledge into the learning process. Here, since our focus is on the unconstrained multi-agent domain, where domain knowledge, when it exists, will more than likely be vague and incomplete, we developed a mechanism to incorporate 'fuzzy' domain knowledge into multi-agent reinforcement learning. Here fuzzy domain knowledge is domain knowledge which is written as a series of fuzzy-logic based rules. Our approach centered on using reward shaping methods similar to the work done by Mataric [90]. We demonstrated our integration mechanism on a simple, training-example constrained pole balancing problem where the learner was given various (and often inadequate) numbers of training examples. We saw how the sample complexity of the learning process was reduced by the integration of some simple fuzzy rules. When performing reward shaping, one must be careful not to so distort the reward function that the learner's post-training performance is harmed. We developed optimality bounds which determined how much potential harm to post-training performance can be done by a given amount of reward shaping. We also demonstrated that only a minimal amount of reward shaping needs to be done to see significant reductions in the sample complexity of the RL process in many domains. We pointed out that this was because domain knowledge translates

into rewards over shorter timescales than typical rewards for reinforcement learning problems, which allows much smaller magnitude domain knowledge rewards to have a large impact of system dynamics.

The use of continuous action spaces can greatly reduce the action space of a multi-agent reinforcement learning problem. In domains where the agent needs access to a highly-segmented action space (for example, an agent aiming a projectile might need an action space corresponding to various targeting angles ($\theta = 0, \theta = .1, \theta = 0.2$,etc.), a performance improvement can often be achieved by using one continuous action space instead of a highly segmented action space with many discrete actions. However, we showed how using continuous action spaces with approximate reinforcement learning can lead to action-selection problems in what we termed "Narrow-Q" domains. Narrow-Q domains arise when reward is distributed very sparsely temporally. We proposed a method to overcome the action-selection problems on Narrow-Q domains by normalizing the Q-function. We bounded the amount of error that can be introduced by this normalization process, and showed how this error can be minimized by an appropriate choice of action-space basis functions.

Finally, we presented an implementation of approximate reinforcement learning which ran almost entirely on the Graphics Processing Unit. We demonstrated how one can parallelize certain aspects of the approximate RL algorithm to achieve a performance increase on the GPU's SIMD-based architecture. We managed to achieve almost a 4.5x speedup over a CPU-based implementation of the approximate RL algorithm, which came close to saturating the maximum possible speedup given the architecture.

A. Final Thoughts

Almost all the research in this dissertation employed, in some way, approximation to reduce the computational complexity of multi-agent reinforcement learning. Whether this was in the form the greedy algorithm we used for information-based coalition formation (which was not guaranteed to maximize coalition information), or the reward modification techniques used to integrate domain knowledge (which potentially harmed post-training performance), or the Q-function normalization techniques used to permit the use of continuous action spaces on narrow-Q domains (which also potentially harmed post-training performance), most of our techniques involved some sort of approximation-based approach. Using approximation like this, in order to overcome the computational complexity of multi-agent reinforcement learning (at least in the unconstrained domain) is necessary, and we believe that the trend for research in this area will increasingly move toward approximation-based approaches. We believe no fundamental computational complexity reduction for either value-iteration or policy-iteration is on the horizon - in order for an RL algorithm to scale polynomially with the number of system agents, the algorithm would have to scale logarithmically with the size of the action space of the system, which seems unlikely.

Additionally, parallel computational offers another avenue for speed improvements to MARL computations. We demonstrated a simple version of this when we performed MARL on the GPU, but much higher performing implementations of parallel MARL could no doubt be achieved (for example, by employing multiple GPUs or other speciality hardware).

In the end, we believe that improvement in the MARL research area will come from approaches other than a direct breakthrough in the complexity of value or policy iteration, and it will be exciting to watch the literature as these alternative research

directions are pursued.

REFERENCES

[1] M. Ljungberg, A. Lucas, The OASIS air-traffic management system, in: Proceedings of the Second Pacific Rim International Conference on Artificial Intelligence (PRICAI), 1992, pp. 411–417.

[2] P. Maes, Agents that reduce work and information overload, Communications of the ACM 37 (7) (1994) 31–40.

[3] A. Chavez, P. Maes, KABASH: An agent marketplace for buying and selling goods, in: Proceedings of First International Conference on the Practical Application of Intelligent Agents and Multi-Agent Systems, 1996, pp. 99–104.

[4] N. Jennings, P. Faratin, M. Johnson, T. Norman, P. O'BRien, M. Wiegand, Agent-based business process management, International Journal of Cooperative Information Systems 5 (2,3) (1996) 105–130.

[5] B. Hayes-Roth, M. Hewett, R. Washington, R. Hewett, A. Seiver, Distributing intelligence within an individual, in: L. Gasser, M. Huhns (Eds.), Distributed Artificial Intelligence: vol. 2, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1990, pp. 385–412.

[6] J. Huang, N. Jennings, J. Fox, An agent-based approach to health care management, Int. Journal of Applied Artificial Intelligence 9 (4) (1995) 401–420.

[7] P. Wavish, M. Graham, A situated action approach to implementing characters in computer games, International Journal of Applied Artificial Intelligence 10 (1) (1996) 53–74.

[8] J. Anderson, The Atomic Components of Thought, Erlbaum, Mahwah, NJ, 1988.

[9] J. Anderson, D. Bothell, M. Byrne, S. Douglass, C. Lebiere, , Y. Qin, An integrated theory of the mind, Psychological Review 111 (4) (2004) 1036–1060.

[10] J. Laird, A. Newell, P. Rosenbloom, SOAR: An architecture for general intelligence, Artificial Intelligence 33 (1987) 1–64.

[11] P. Stone, M. Veloso, Team-partitioned, opaque-transition reinforcement learning., in: Proceedings of the Third International Conference on Autonomous Agents (ICAA 1999), 1999, pp. 135–140.

[12] S. Post, M. Fassaert, A. Visser, The high-level communication model for multi-agent coordination in the robocuprescue simulator, Lecture Notes on Artificial Intelligence 3020 (2004) 503–509.

[13] A. Fraser, B. Donald, Simulation of genetic systems by automatic digital computers, Australian Journal of Biological Sciences 10 (1957) 484–491.

[14] D. B. F. (editor), Evolutionary Computation: The Fossil Record, IEEE Press, New York, 1998.

[15] J. Crosby, Computer Simulation in Genetics, John Wiley & Sons, New York, 1973.

[16] D. Rumelhart, J. McClelland, Parallel Distributed Processing: Explorations in the Microstructure of Cognition, The MIT Press, Cambridge, MA, 1986.

[17] R. Sutton, A. Barto, Reinforcement Learning: An Introduction, MIT Press, Cambridge, MA, 1998.

[18] L. Kaelbling, M. Littman, A. Moore, Reinforcement learning: A survey, Journal of Artificial Intelligence Research 4 (1996) 237–285.

[19] D. Ernst, S. Guy-Bart, J. Goncalves, L. Wehenkel, Clinical data-based optimal STI strategies for HIV: A reinforcement learning approach, in: Proceedings of Beneleard 2006, Ghent, Belgium, 2006, pp. 65–72.

[20] M. Walker, An application of reinforcement learning to dialogue strategy selection in a spoken dialogue system for e-mail, Journal of Artificial Intelligence Research 12 (2000) 387–416.

[21] D. Ernst, R. Maree, L. Wehenkel, Reinforcement learning with raw image pixels as state input, in: Lecture Notes in Computer Science, Volume 4153, 2006.

[22] M. Lagoudakis, R. Parr, Model-free least-squares policy iteration, Advances in Neural Information Processing Systems 14 (2002) 65–90.

[23] M. Lagoudakis, R. Parr, Least-squares policy iteration, Journal of Machine Learning Research 4 (2003) 1107–1149.

[24] M. G. Lagoudakis, R. Parr, Least-squares policy iteration, Journal of Machine Learning Research 4 (6) (2004) 1107–1150.

[25] R. Coulom, Reinforcement learning using neural networks, with applications to motor control, Ph.D. thesis, Institut National Polytechnique de Grenoble, France (2002).

[26] B. Marthi, S. Russell, D. Latham, C. Guestrin, Concurrent hierarchical reinforcement learning, in: Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI 2005), Edinburgh, Scotland, 2005, pp. 310–317.

[27] A. Pouget, C. Deffayet, T. Sejnowski, Reinforcement learning predicts the site of plasticity for auditory remapping in the barn owl, in: D. Tesauro, D. Touretzky, J. Alspector (Eds.), Advances in Neural Information Processing Systems, MIT Press, Cambridge, MA, 1995, pp. 125–132.

[28] F. Worgotter, Temporal sequence learning, prediction, and control: A review of different models and their relation to biological mechanisms, Neural Computation 17 (2005) 217–230.

[29] R. Bellman, Dynamic Programming, Princeton University Press, Princeton, NJ, 1957.

[30] R. Howard, Dynamic Programming and Markov Processes, Massachusetts Institute of Technology Press, Cambridge, MA, 1960.

[31] J. Rust, Numerical dynamic programming in economics, in: H. Amman, D. Kendrick, J. Rust (Eds.), Handbook of Computational Economics, Elsevier, Amsterdam, 1996, pp. 614–722.

[32] D. Bertsekas, Distributed dynamic programming, IEEE Transactions on Automatic Control 27 (1982) 610–616.

[33] D. Bertsekas, Distributed asynchronous computation of fixed points, Mathematical Programming 27 (1983) 107–120.

[34] W. Lovejoy, A survey of algorithmic methods for partially observable Markov decision processes, Annals of Operations Research 28 (1991) 47–66.

[35] C. Watkins, P. Dayan, Q-learning, Machine Learning 8 (3) (1992) 279–292.

[36] C. Watkins, Learning from delayed rewards, Ph.D. thesis, King's College, Cambridge (1989).

[37] M. Minsky, Theory of neural-analog reinforcement systems and its application to the brain-model problem, Ph.D. thesis, Princeton University, Princeton, NJ (1954).

[38] D. Hebb, The Organization of Behavior, Wiley, New York, 1949.

[39] B. Farley, W. Clark, Simulation of self-organizing systems by digital computer, IRE Transactions on Information Theory 4 (1954) 76–84.

[40] M. Minsky, Steps toward artificial intelligence, Proceedings of the Institute of Radio Engineers 49 (1961) 8–30.

[41] D. Michie, Trial and error, in: S. Barnett, A. McLaren (Eds.), Science Survey, Part 2, Penguin, Harmondsworth, London, 1961, pp. 129–145.

[42] D. Michie, Experiments on the mechanisation of game learning: Characterization of the model and its parameters., Computer Journal 1 (1963) 232–263.

[43] D. Michie, R. Chambers, Boxes: An experiment in adaptive control, in: E. Dale, D. Michie (Eds.), Machine Intelligence 2, 1968, pp. 127–152.

[44] B. Widrow, N. Gupta, S. Maitra, Punish/reward: Learning with a critic in adaptive threshold systems, IEEE Transactions on Systems, Man, and Cybernetics 5 (1973) 455–465.

[45] A. Samuel, Some studies in machine learning using the game of checkers, IBM Journal of Research and Development 3 (1959) 210–229.

[46] G. Tesauro, Practical issues in temporal difference learning, Machine Learning 8 (1992) 257–278.

[47] D. P. Bertsekas, Dynamic Programming: Deterministic and Stochastic Models, Prentice-Hall, Englewood Cliffs, NJ, 1987.

[48] D. P. Bertsekas, J. N. Tsitsiklis, Parallel and Distributed Computation – Numerical Methods, Prentice Hall, Cambridge, MA, 1989.

[49] M. Littman, T. Dean, L. Kaelbling, On the complexity of solving Markov decision problems, in: Proceedings of the 11th Annual Conference on Uncertainty in Artificial Intelligence (UAI-95), Morgan Kaufmann, San Francisco, CA, 1995, pp. 394–400.

[50] M. Puterman, Markov Decision Processes - Discrete Stochastic Dynamic Programming, John Wiley & Sons, Inc., New York, 1994.

[51] T. Jaakkola, M. Jordan, S. Singh, On the convergence of stichastic iterative dynamic programming algorithms, Neural Computation 6 (6) (1994) 56–70.

[52] J. Tsitsiklis, Asynchronous stochastic approximation and q-learning, Machine Learning 16 (3) (1994) 29–36.

[53] D. Wolpert, K. Wheeler, K. Turner, General principles of learning-based mutliagent systems, in: International Conference on Autonomous Agents, 1999, pp. 2–9.

[54] M. Tan, Multi-agent reinforcement learning: Independent vs. cooperative agents., in: Proceedings of the Tenth International Conference on Machine Learning, 1993, pp. 330–337.

[55] J. Hu, M. Wellman, Multiagent reinforcement learning: Theoretical framework and an algorithm, in: International Conference on Machine Learning (ICML 1998), pp. 242–250.

[56] M. Bowling, M. Veloso, Multi-agent learning using a variable learning rate, Artificial Intelligence 136 (2002) 215–250.

[57] M. Bowling, M. Veloso, Simultaneous adversarial multi-robot learning, in: Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence, 2003, pp. 13–18.

[58] S. Kakade, A natural policy gradient, in: Proceedings of the Fourteenth Annual Conference on Neural Information Processing Systems (NIPS14), 2002, pp. 89–94.

[59] D. Guestrin, D. Koller, R. Parr, Multiagent planning with factored MDPs, in: Proceedings of the Fourteenth Annual Conference on Neural Information Processing Systems (NIPS14), 2001, pp. 116–122.

[60] C. Guestrin, D. Koller, R. Parr, Max-norm projections for factored MDPs, in: Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence, 2001, pp. 113–119.

[61] C. Guestrin, M. Lagoudakis, R. Parr, Coordinated reinforcement learning, in: Machine Learning: Proceedings of the Nineteenth International Conference (ICML 2002), 2002, pp. 322–328.

[62] R. P. S. V. C. Guestrin, D. Koller, Efficient solution algortihms for factored MDPs, Journal of Artificial Intelligence Research 19 (2003) 399–468.

[63] O. Shehory, S. Kraus, Task allocation via coalition formation among autonomous agents, in: Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI 1995), 1995, pp. 277–285.

[64] O. Shehory, S. Kraus, Methods for task allocation via agent coalition formation, Artificial Intelligence Journal 101 (1,2) (1998) 165–200.

[65] S. Sen, P. Dutta, Searching for optimal coalition structures, in: Fourth International Conference on Multi-Agent Systesm (ICMAS 2000), 2000, pp. 287–300.

[66] T. Sandholm, K. Larson, M. Anderson, O. Shehory, F. Tohme, Coalition structure generation with worst case guarantees, Artificial Intelligence 111 (1,2) (1999) 209–238.

[67] J. Schneider, W. Wong, A. Moore, M. Riedmiller, Distributed value functions, International Conference on Machine Learning (ICML 1999) (1999) 441–448.

[68] M. Tsvetovat, K. Sycara, Customer coalitions in the electronic marketplace., in: Proceedings of the Fourth International Conference on Autonomous Agents, 2000, pp. 263–264.

[69] I. Foster, C. Kesselman, S. Tuecke, The anatomy of the grid, The International Journal of High Performance Computing Applications 15 (3) (2001) 200–222.

[70] T. Norman, A. Preece, Agent-based formation of virtual organizations, International Journal of Knowledge Based Systems 17 (2-4) (2004) 103–111.

[71] V. Dang, N. Jennings, Generating coalition structures with finite bound from the optimal guarantees, in: Proceedings of the 3rd International Conference on Autonomous Agents and Multi-Agent Systems, 2004, pp. 563–571.

[72] R. Battiti, Using mutual information for selecting features in supervised neural net learning, IEEE Transactions on Neural Networks 5 (4) (1994) 537–550.

[73] T. Jaakkola, M. Meila, T. Jebara, Maximum entropy discrimination, Advances in Neural Information Processing Systems 12 (1999) 415–440.

[74] R. Kohavi, Wrappers for performance enhancement and oblivious decision graphs, Ph.D. thesis, Stanford University, Palo Alto, CA (1995).

[75] W. Duch, J. Biesiade, T. Winiarski, K. Grudzinski, K. Grabczewski, Feature selection based on information theory filters and feature elimination wrapper methods, in: Proceedings of the Int. Conf. on Neural Networks and Soft Computing (INNSC 2002), 2002, pp. 311–316.

[76] P. Devijver, J. Kittler, Pattern Recognition: A Statistical Approach, Prentice Hall, Cambridge, MA, 1982.

[77] C. Bishop, Neural Networks for Pattern Recognition, Oxford University Press, Oxford, UK, 1996.

[78] H. Ragavan, L. Rendell, Lookahead feature construction for learning hard concepts, in: International Conference on Machine Learning (ICML 1993), 1993, pp. 252–259.

[79] S. Zhang, L. Xie, M. Adams, Entropy based feature selection scheme for real-time simultaneous localization and map building, in: Intelligent Robots and Systems 2005 (IROS 2005), 2005, pp. 1–12.

[80] R. Setiono, H. Liu, Improving backpropagation learning with feature selection, Neural Networks and Complex Problem-Solving Technologies 6 (1996) 129–139.

[81] O. Shehory, S. Kraus, Formation of overlapping coalitions for precendence-ordered task-execution among autonomous agents, in: Proceedings of International Conference on Multi-Agent Systems (ICMAS 1996), Kyoto, Japan, 1996, pp. 330–337.

[82] C. Shannon, A mathematical theory of communication, Bell System Technical Journal 27 (1948) 623–656.

[83] G. Towell, J. Shavlik, Knowledge-based artificial neural networks, Artificial Intelligence 70 (1994) 12–70.

[84] S. Mahadevan, J. Connell, Automatic programming of behavior-based robots using reinforcement learning, Artificial Intelligence 55 (1992) 311–365.

[85] J. Millan, Rapid, safe, and incremental learning of navigation strategies, IEEE Transactions on Systems, Man and Cybernetics: Part B. Special Issue on Learning Autonomous Robots 26 (3) (1996) 408–420.

[86] Y. Abu-Mostafa, Hints, Neural Computation 7 (4) (1995) 639–671.

[87] Y. Abu-Mostafa, Financial model calibration using consistency hints, IEEE Transactions on Neural Networks 12 (4) (2001) 791–808.

[88] Y. Abu-Mostafa, Learning from hints in neural networks, Journal of Complexity 6 (2) (1990) 192–198.

[89] J. Sill, Y. Abu-Mostafa, Monotonicity hints for credit screening, in: Proceedings of the International Conference on Neural Information Processing (ICONIP'96), 1996, pp. 123–127.

[90] M. Mataric, Reward functions for accelerated learning, in: Proceedings of the Eleventh International Conference on Machine Learning, 1994, pp. 8–16.

[91] A. Y. Ng, D. Harada, S. Russell, Policy invariance under reward transformations: theory and application to reward shaping, in: Proceedings of the 16th International Conference on Machine Learning, Morgan Kaufmann, San Francisco, CA, 1999, pp. 278–287.

[92] G. Hailu, G. Sommer, Embedding knowledge in reinforcement learning, in: International Conference on Artificial Neural Networks (ICANN), 1998, pp. 1133–1138.

[93] L. Zadeh, Fuzzy sets, Information and Control 8 (1965) 338–353.

[94] L. Zadeh, Fuzzy algorithms, Information and Control 5 (1968) 94–102.

[95] D. Bertsekas, J. Tsitsiklis, Neuro-Dynamic Programming, Athena Scientific, Nashua, NH, 1996.

[96] C. Karr, Design of a cart-pole balancing fuzzy logic controller using a genetic algorithm, in: Proceedings of The International Society for Optical Engineering Vol. 1468, 1991, pp. 26–36.

[97] V. Cerny, Thernodynamical approach to the traveling salesman problem: An efficient simulation algorithm, Journal of Optimization Theory and Applications 1 (1985) 41–51.

[98] S. Kirkpatrick, C. Gelatt, M. Vecchi, Optimization by simulated annealing, Science 220 (4598) (1983) 671–680.

[99] N. Metropolis, A. Rosenbluth, A. Teller, E. Teller, Equations of state calulations by fast computing machines, Journal of Chemical Physics 21 (6) (1953) 1087–1092.

[100] Y. Nourani, B. Andersen, A comparison of simulated annealing cooling strategies, Journal of Physics A: Mathematical and General 31 (41) (1998) 8373–8385.

[101] E. Aarts, P. Laarhoven, A new polynomial time cooling schedule, in: Proceedings of the IEEE International Conference on Computer Aided Design, 1985, pp. 206–208.

[102] M. Huang, F. Romeo, A. Sangiovanni-Vincentelli, An efficient general cooling schedule for simulated annealing, in: Proceedings of the IEEE International Conference on Computer Aided Design, 1986, pp. 381–384.

[103] C. Ernoult, Performance of backpropagation on a parallel transputer-based machine, in: Proceedings of the International Joint Conference on Artificial Intelligence, Morgan Kaufmann, San Francisco, CA, 1987, pp. 323–326.

[104] R. Kretchmar, Parallel reinforcement learning, in: SCI2002. The 6th World Conference on Systemics, Cybernetics, and Informatics, 2002, pp. 165–170.

[105] A. Likas, K. Blekas, A. Stafylopatis, Parallel recombinative reinforcement learning: A genetic approach, Journal of Intelligent Systems 6 (2) (1996) 145–170.

[106] M. Rumpf, R. Strzodka, Using graphics cards for quantized fem computations, in: Proceedings of IASTED Visualization, Imaging and Image Processing Conference (VIIP'01), 2001, pp. 193–202.

[107] T. Kim, M. Lin, Visual simulation of ice crystal growth, in: Proceedings of the ACM SIGGARPH / Eurographics Symposium on Computer Animation, 2003, pp. 111–116.

[108] M. Harris, W. Baxter, T. Scheuermann, A. Lastra, Simulation of cloud dynamics on graphics hardware, in: HWWS '03: Proceedings of the ACM SIGGARPH/EUROGAPRHICS conference on Graphics hardware, Eurographics Association, Aire-la-Ville, Switzerland, 2003, pp. 92–101.

[109] Z. Fan, F. Qiu, A. Kaufman, S. Yoakum-Stover, GPU cluster for high performance computing, in: ACM / IEEE Supercomputing Conference, 2004, pp. 211–216.

[110] J. Bolz, I. Farmer, E. Grinspun, P. Schroder, Sparse matrix solvers on the GPU: Conjugate gradients and multigrid, ACM Transactions on Graphics 22 (3) (2003) 917–924.

[111] J. Kruger, R. Westermann, Linear algebra operators for GPU implementation of numerical algorithms, ACM Transactions on Graphics 22 (3) 908–916.

[112] D. Goddeke, GPGPU performance tuning, Tech. rep., University of Dortmund, http://www.mathematik.uni-dortmund.de/ goeddeke/gpgpu/ (2005).

[113] A. Lastra, M. Lin, D. Manocha, Survey of GPGPU applications, in: Proceedings of the ACM Workshop on General Purpose Computation on Graphics Processors, 2004, pp. 123–130.

[114] N. Galoppo, N. Govindaraju, M. Henson, D. Manocha, LU-GPU: Efficient algorithms for solving dense linear systems on graphics hardware, in: Proceedings of the ACM/IEEE SC'05 Conference, 2005, pp. 411–417.

[115] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Mike, H. Pat, Brook for GPUs: Stream computing on graphics hardware (2004).
URL `citeseer.ist.psu.edu/buck04brook.html`

[116] R. Fernando, GPGPU: General-purpose computation on GPUs, NVIDIA Developer Technology Group Documentation (2005).

APPENDIX A

CODE LISTINGS

File: IBCF Psudocode

```
Set MAX_COALITION_SIZE = X Set CS = {} Set POOL = {n1,n2,...,nN}

GENERATE TRAINING SAMPLES FOR AGENTS IN POOL While POOL Not Empty

    Set C = {}

    Set IMPROVED = True

    While IMPROVED = True AND |C| < MAX_COALITION_SIZE

      IMAX = I(C)

      IC = I(C)

      IMPROVED = False

        For i = 1 to |POOL|

            C' = {C,ni}

            IC' = I(C')

            If (IC' > IMAX)

                nMAX = ni

                IMAX = IC'

            End

        End

        If IMAX > IC
```

```
            IMPROVED = True

            C = C + nMAX

            CS = {CS, C}

            POOL = POOL - nIndex

        End

    End

    CS = CS + C

End
```

File: main_program.br

```
#include <stdio.h> #include <math.h>

void resetTimer(void); float getTimer(void);

//**************************************************************************

    Although no more than 2048 samples could be processed on this

    graphics card, we built in support for multiple 'runs'. Here

    we've set this number to 40, allowing for many more samples

    to be processed than would fit in a single row vector stream.
```

```
***********************************************************************//

#define NRUN            40

#define SAMPLES         2000

#define NRUN_SAMPLES        80000

// discounting parameter GAMMA ... used in LSPI

#define GAMMA           0.9f

// the number of basis functions used for approximation

#define BASIS_FUNCTIONS     10

// the number of state variables

#define VARS            3

// the number of possible actions

#define ACTIONS         4 #define ACTION_BASIS_FUNCTIONS  40

// when computing A, we may want to compute several columns at once

// GROUP defines how many columns of A we should compute simultaneously
```

```
#define GROUP               8

#define ACTION_BASIS_FUNCTIONS_GROUP 320




// data structure for the experience tuples

// this will be used for the pre and post information streams


typedef struct Tuple_t {


  float x1;

  float x2;

  float action;

  float reward;


} Tuple;


// basis seed information ...

typedef struct BasisSeed_t {

   float x1;

   float x2;

   float action;
```

```
} BasisSeed;



// kernel to compute the basis functions ... with the delta function
// part considered

kernel void basis_function( Tuple data<>,  BasisSeed seed<>, out
float basis<>) {
    float c1,c2;

    {
        {
            c1 =    data.x1 - seed.x1;
            c2 = data.x2 - seed.x2;

            basis = exp( - (c1*c1 + c2*c2) / 3 );
        }

        // if this basis functions was used for bias
        // just set it to 1
        if (seed.x1 <= -100)
            basis = 1;
    }
```

```
    // check if the delta function zeros out the basis function
    if (seed.action != data.action)
        basis = 0;
}

// kernel to compute the basis functions ... with the delta function
// part NOT considered

kernel void basis_function_all( Tuple data<>,  BasisSeed seed<>, out
float basis<>) {
    float c1,c2;

    {
        c1 = data.x1 - seed.x1;
        c2 = data.x2 - seed.x2;
        basis = exp( - (c1*c1 + c2*c2) / 3 );
    }

    // if this basis function was used for bias
    // just set it to 1
    if (seed.x1 < -100)
        basis = 1;
```

```
}

//simple copy kernel
kernel void copy (float source<>, out float target<>) {
    target = source;
}

//simple copy and then add kernel
kernel void copy_add (float source<>, float source2<>, out float
target<>) {
    target = source + source2;
}

//kernel useful in computing the Z matrix
kernel void copy_add_mod(float source<>, float source2<>, float mod,
out float target<>) {
    target = source + mod *source2;
}

//basis sum reduction
reduce void sum ( float data<>, reduce float output<> ) {
    output += data;
```

```
}

//same thing, except reduce to a single float
reduce void fsum ( float data<>, reduce float output ) {
    output += data;
}

//simple max reduction
reduce void max (float source<>, reduce float output<>) {
    if (source > output)
        output = source;
}

//simple multiplication kernel
kernel void mult(float one<>, float two<>, out float output<>) {
    output = one * two;
}

//this kernel finds the index of the maximum Qa values
kernel void find_action_max (float input<>, float index<>, float
action_max<>, out float action<>) {
    if (input < action_max)
        action = 0;
```

```
    else
        action = index;
}

//this kernel is used to put the 'post' actions into the post information stream
kernel void replace_actions (Tuple input<>, float actions<>, out
Tuple output<>) {
    output.x1 = input.x1;
    output.x2 = input.x2;
    output.action = actions;
}


#define data raw.domain(int2(0,b),int2(SAMPLES,b+1)) #define pdata
praw.domain(int2(0,b),int2(SAMPLES,b+1))

main() {

  //basic index variables
  int i,j,b,index;

  {
```

```
// the 'pre' information stream
Tuple   *input_raw;
Tuple   raw<NRUN,SAMPLES>;

// the 'post' information stream
Tuple   *input_praw;
Tuple   praw<NRUN,SAMPLES>;

// used to hold basis functions for the 'pre' and 'post'
// basis function information streams X and Y
float   basis< ACTION_BASIS_FUNCTIONS, SAMPLES >;
float   pbasis< ACTION_BASIS_FUNCTIONS, SAMPLES >;

// used to hold the Q stream
float   qvalues< ACTION_BASIS_FUNCTIONS, SAMPLES>;

// basis seed streams
BasisSeed   input_seed[ACTION_BASIS_FUNCTIONS];
BasisSeed   seed< ACTION_BASIS_FUNCTIONS, 1>;

// various intermediate streams used in policy evaluation/action selection
float   action_temp< ACTIONS, SAMPLES >;
float   action_max<1, SAMPLES >;
```

```
float   action_index<ACTIONS, 1>;

float   action_index_input[ACTIONS] = {1,2,3,4};


// stream used to store omega

float   input_coef[ACTION_BASIS_FUNCTIONS];

float   coef<ACTION_BASIS_FUNCTIONS,1>;


// used to hold the actual A matrix

float   A<ACTION_BASIS_FUNCTIONS, ACTION_BASIS_FUNCTIONS>;


// used to output the A matrix to the screen

float   output_A[ACTION_BASIS_FUNCTIONS][ACTION_BASIS_FUNCTIONS];


// timer

float   time;


// temporary matrix used to compute A

float   dot_space<ACTION_BASIS_FUNCTIONS_GROUP,SAMPLES>;

float   Asum_space<ACTION_BASIS_FUNCTIONS_GROUP,1>;


// indexes used to handle batch processing of the columns of A

int k,store_col[GROUP];
```

```
// allocate the 'pre' and 'post' information streams
input_raw = (Tuple*)malloc( NRUN_SAMPLES*sizeof(Tuple) );
input_praw = (Tuple*)malloc( NRUN_SAMPLES*sizeof(Tuple) );

// initialize the 'post' actions to zero
streamRead(action_index,action_index_input);

//read-in the training data
FILE *fp;

// read in 'pre' state information

fp = fopen("c:\\x1_arc.txt","r+");
for (i = 0;i<NRUN_SAMPLES;i++)
    fscanf(fp, "%f", &input_raw[i].x1);
fclose(fp);

fp = fopen("c:\\x2_arc.txt","r+");
for (i = 0;i<NRUN_SAMPLES;i++)
        fscanf(fp, "%f", &input_raw[i].x2);
fclose(fp);
```

```
// read in 'post' state information

fp = fopen("c:\\x1p_arc.txt","r+");
for (i = 0;i<NRUN_SAMPLES;i++)
    fscanf(fp, "%f", &input_praw[i].x1);
fclose(fp);

fp = fopen("c:\\x2p_arc.txt","r+");
for (i = 0;i<NRUN_SAMPLES;i++)
    fscanf(fp, "%f", &input_praw[i].x2);
fclose(fp);

// read in 'pre' action information
fp = fopen("c:\\action_arc.txt","r+");
for (i = 0;i<NRUN_SAMPLES;i++)
    fscanf(fp, "%f", &input_raw[i].action);
fclose(fp);

// read in the initial omega
fp = fopen("c:\\coef.txt","r+");
for (i=0;i<ACTION_BASIS_FUNCTIONS;i++)
    fscanf(fp,"%f",&input_coef[i]);
fclose(fp);
```

```
// read in the basis seed information
fp = fopen("c:\\seed_data.txt","r+");
for (i=0;i<ACTION_BASIS_FUNCTIONS;i++)
   fscanf(fp, "%f, %f, %f", &input_seed[i].x1,
            &input_seed[i].x2,&input_seed[i].action);
fclose(fp);



//transfer all this information to video memory


streamRead( coef, input_coef );
streamRead( raw, input_raw );
streamRead( praw, input_praw );
streamRead( seed, input_seed );



//zero out the matrix A
for (i=0;i<ACTION_BASIS_FUNCTIONS;i++)
    for (j=0;j<ACTION_BASIS_FUNCTIONS;j++)
        output_A[i][j]=0;


streamRead( A, output_A );
```

```
//start the timer
resetTimer();

// do this for each run
for (b=0;b<NRUN;b++)
{

//compute the basis functions (in the dissertation, this is computing F)
basis_function_all( data, seed, basis );

//find the actions produced by the current policy
mult( coef, basis, qvalues );
sum( qvalues, action_temp);
max( action_temp, action_max);
find_action_max(action_temp, action_index, action_max, action_temp);
sum(action_temp, action_max);

//fill in the 'post' action fields in the
//'post' information stream
replace_actions( pdata,action_max,pdata);

//compute the basis functions (in the dissertation, computes X and Y)
```

```
basis_function( data, seed, basis );
basis_function( pdata, seed, pbasis );

//compute the Z intermediate matrix
copy_add_mod(basis,pbasis,-0.9f,pbasis);

//compute the matrix A
index = 0;k=0;

//for each of the columns of A
for (i=0;i<ACTION_BASIS_FUNCTIONS;i++)
{
        //perform the multiplication part of the dot product
        mult(basis, pbasis.domain(int2(0,i),int2(SAMPLES,i+1)),
           dot_space.domain(int2(0,ACTION_BASIS_FUNCTIONS*index),
                        int2(SAMPLES,ACTION_BASIS_FUNCTIONS*(index+1)))
                   );

        //keep track that we have another column of A ready for processing
        store_col[index]=i;
            index++;

        //if we've got enough ready in this batch of A columns ...
```

```
if (index >= GROUP)
{
       // perform the summation on the entire batch at once


     sum(dot_space.domain(
                int2(0,ACTION_BASIS_FUNCTIONS*k),
                int2(SAMPLES,ACTION_BASIS_FUNCTIONS*(k+1))),
        Asum_space);


   //now copy the columns into the appropriate place in video memory
   // ... copy them to where A is stored
    for (k=0;k<GROUP;k++)
    {


        copy_add(


            A.domain(int2(store_col[k],0),
            int2(store_col[k]+1,ACTION_BASIS_FUNCTIONS)),


            Asum_space.domain(int2(0,ACTION_BASIS_FUNCTIONS*k),
            int2(1,ACTION_BASIS_FUNCTIONS*(k+1))),


            A.domain(int2(store_col[k],0),
```

```
                    int2(store_col[k]+1,ACTION_BASIS_FUNCTIONS))

                    );
            }

        index = 0;
        }

}
}

// stop timer
time = getTimer();

// write our results to a file
fp = fopen("c:\\output.txt","w+");
fprintf(fp, "%f",time);
fclose(fp);

//print A on the screen
streamWrite( A, output_A );

for(j=0;j<5;j++)
```

```
    {
        for (i=0;i<5;i++)
    {
            fprintf(stderr, " %2.3f ",output_A[i][j]);
    }

    fprintf(stderr,"\n");
    }

    fprintf(stderr,"\n\n Time: %f \n\n", time);

}

  return 0;
}
```

File: basis_seed.txt

```
 -200, 0, 1
```

```
-1,-1, 1 -1, 0, 1 -1, 1, 1
 0,-1, 1
 0, 0, 1
 0, 1, 1
 1,-1, 1
 1, 0, 1
 1, 1, 1
 -200, 0, 2
-1,-1, 2 -1, 0, 2 -1, 1, 2
 0,-1, 2
 0, 0, 2
 0, 1, 2
 1,-1, 2
 1, 0, 2
 1, 1, 2
 -200, 0, 3
-1,-1, 3 -1, 0, 3 -1, 1, 3
 0,-1, 3
 0, 0, 3
 0, 1, 3
 1,-1, 3
 1, 0, 3
 1, 1, 3
```

```
-200, 0, 4 -1,-1, 4 -1, 0, 4 -1, 1, 4
 0,-1, 4
 0, 0, 4
 0, 1, 4
 1,-1, 4
 1, 0, 4
 1, 1, 4
```

VITA

Name: Victor Palmer

Address: 3313 19th Street, Lubbock TX., 79410

Email Address: vpalmer@gmail.com

Education: B.S., Mathematics, Lubbock Christian University 1998.