AAAI Spring Symposium, Stanford, California, March 2002.

Coordinated Reinforcement Learning

Carlos Guestrin
Computer Science Dept
Stanford University
guestrin@cs.stanford.edu

Michail Lagoudakis
Computer Science Dept
Duke University
mgl@cs.duke.edu

Ronald Parr
Computer Science Dept
Duke University
parr@cs.duke.edu

Abstract

We present several new algorithms for multiagent reinforcement learning. A common feature of these algorithms is a parameterized, structured representation of a policy or value function. This structure is leveraged in an approach we call *coordinated reinforcement learning*, by which agents coordinate both their action selection activities and their parameter updates. Within the limits of our parametric representations, the agents will determine a jointly optimal action without explicitly considering every possible action in their exponentially large joint action space. Our methods differ from many previous reinforcement learning approaches to multiagent coordination in that structured communication and coordination between agents appears at the core of both the learning algorithm and the execution architecture.

1 Introduction

Consider a system where multiple agents, each with its own set of possible actions and its own observations, must coordinate in order to achieve a common goal. We want to find a mechanism for coordinating the agents' actions so as to maximize their joint utility. One obvious approach to this problem is to represent the system as a Markov decision process (MDP), where the "action" is a joint action for all of the agents and the reward is the total reward for all of the agents. The immediate difficulty with this approach is that the action space is quite large: If there are g agents, each of which can take a actions, then the action space is a^g.

One natural approach to reducing the complexity of this problem is to restrict the amount of information that is available to each agent and hope to maximize global welfare by solving local optimization problems for each agent [14]. In some cases, it is possible to manipulate the presentation of information to the agents in a manner that forces local optimizations to imply global optimizations [17]. In general, however, the problem of finding a globally optimal solution for agents with partial information is known to be intractable [2].

Following [7] we present an approach that combines value function approximation with a message passing scheme by which the agents efficiently determine the jointly optimal action with respect to an approximate value function. Our approach is based on approximating the joint value function as a linear combination of local value functions, each of which relates only to the parts of the system controlled by a small number agents. We show how such factored value functions allow the agents to find a globally optimal joint action using a very natural message passing scheme. This scheme can be implemented as a negotiation procedure for selecting actions at run time. Alternatively, if the agents share a common observation vector, each agent can efficiently determine the actions that will be taken by all of the collaborating agents without any additional communication.

Given an action selection mechanism, the remaining task is to develop a reinforcement learning algorithm that is capable of producing value functions of the appropriate form. An algorithm for computing such value functions is presented in [7] for the case where the model is known and represented as a factored MDP. This is the first application of these techniques in the context of reinforcement learning, where we no longer require a factored model or even a discrete state space.

We begin by presenting two methods of computing an appropriate value function through reinforcement learning, a variant of Q-learning and a variant of model free policy iteration called Least Squares Policy Iteration (LSPI) [11]. We also demonstrate how parameterized value functions of the form acquired by our reinforcement learning variants can be combined in a very natural way with direct policy search methods such as [13; 12; 1; 15; 9]. The same communication and coordination structures used in the value function approximation phase are used in the policy search phase to sample from and update a factored stochastic policy function.

We call our overall approach *Coordinated Reinforcement Learning* because structured coordination between agents is used in core of our learning algorithms and in our execution architectures. Our experimental results indicate that, in the case of LSPI, the message passing action selection mechanism and value function approximation can be combined to produce effective policies.

2 Cooperative Action Selection

We begin by considering the simpler problem of having a group of agents select a globally optimal joint action in order to maximize the sum of their individual utility functions. Suppose we have a collection of agents, where each agent j must choose an action a_j from a finite set of possible actions $\mathrm{Dom}(A_j)$. We use $\mathbf{A}$ to denote $\{A_1, \ldots, A_g\}$. The agents are acting in a space described by a set of state variables, $\mathbf{X} = \{X_1 \ldots X_n\}$, where each X_j takes on values in some domain $\mathrm{Dom}(X_j)$. A state $\mathbf{x}$ defines a setting $x_j \in \mathrm{Dom}(X_j)$ for each variable X_j and an action $\mathbf{a}$ defines an action $a_j \in \mathrm{Dom}(A_j)$ for each agent. The agents must choose the joint action $\mathbf{a}$ that maximizes the total utility.

In general, the total utility Q will depend on all state variables $\mathbf{X}$ and on the actions of all agents $\mathbf{A}$. However, in many practical problems, it is possible to approximate the total utility Q by the sum of local sub-utilities Q_j, one for each agent. Now, the total utility becomes $Q = \sum_j Q_j$. For example, consider the decision process of a section manager in a warehouse. Her local utility Q_j may depend on the state of the inventory of her section, on her decision of which products to stock up and on the decision of the sales manager over pricing and special offers. On the other hand, it may not depend directly on the actions of the customer support team. However, the decisions of the support team will be relevant, as they may affect the actions of the sales manager.

Computing the action that maximizes $Q = \sum_j Q_j$ in such problem seems intractable *a priori*, as it would require the enumeration of the joint action space of all agents. Fortunately, by exploiting the local structure in the Q_j functions through a *coordination graph* we can compute the optimal action very efficiently, with limited communication between agents and limited observability, as proposed in [7]. We repeat the construction here as it will be important throughout this paper.

In our framework, each agent j has a local utility function Q_j. An agent's local Q function might be influenced by a subset of the state variables, the agent's action and those of some other agents; we define $\mathrm{Scope}[Q_j] \subset \mathbf{X} \cup \mathbf{A}$ to be the set of state variables and agents that influence Q_j. (We use $Q_j(\mathbf{x}, \mathbf{a})$ to denote the value of Q_j applied to the instantiation of the variables in $\mathrm{Scope}[Q_j]$ within $\mathbf{x}, \mathbf{a}$.) The scope of Q_j can be further divided into two parts: the observable state variables:

$$Observable[Q_j] = \{X_i \in \mathbf{X} \mid X_i \in \mathrm{Scope}[Q_j]\};$$

and the relevant agent decision variables:

$$Relevant[Q_j] = \{A_i \in \mathbf{A} \mid A_i \in \mathrm{Scope}[Q_j]\}.$$

This distinction will allow us to characterize the observations each agent needs to make and the type of communication needed to obtain the jointly optimal action, i.e., the joint action choice that maximizes the total utility $Q = \sum_j Q_j$. We note that each Q_j may be further decomposed as a linear combination of functions that involve fewer variables; in this case, the complexity of the algorithm may be further reduced.

Recall that our task is to find a coordination strategy for the agents to maximize $\sum_j Q_j$ at state $\mathbf{x}$. First, note that the scope of the Q_j functions that comprise the value can include both action choices and state variables. We assume that the agents have full observability of the relevant state variables, i.e., agent j can observe $Observable[Q_j]$. Given a particular state $\mathbf{x} = \{x_1, \ldots, x_n\}$, agent j can instantiate the part of Q_j that depends on the state $\mathbf{x}$, i.e., condition Q_j on state $\mathbf{x}$. Note that each agent only needs to observe the variables in $Observable[Q_j]$, thereby decreasing considerably the amount of information each agent needs to gather.

After conditioning on the current state, each Q_j will only depend on the agent's action choice A_j. Our task is now to select a joint action $\mathbf{a}$ that maximizes $\sum_j Q_j(\mathbf{a})$. The fact that the Q_j depend on the actions of multiple agents forces the agents to coordinate their action choices. We can represent the coordination requirements of the system using a *coordination graph*, where there is a node for each agent and an edge between two agents if they must directly coordinate their actions to optimize some particular Q_i. Fig. 1 shows the coordination graph for an example where

$$Q = Q_1(a_1, a_2) + Q_2(a_2, a_4) + Q_3(a_1, a_3) + Q_4(a_3, a_4).$$

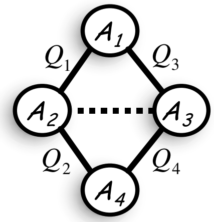

Figure 1: Coordination graph for a 4-agent problem.

A graph structure suggests the use of a *cost network* [6], which can be solved using *non-serial dynamic programming* [3] or a variable elimination algorithm which is virtually identical to variable elimination in a Bayesian network. We review this construction here, as it is a key component in the rest the paper.

The key idea is that, rather than summing all functions and then doing the maximization, we maximize over variables one at a time. When maximizing over a_l, only summands involving a_l participate in the maximization. In our example, we wish to compute:

$$\max_{a_1,a_2,a_3,a_4} Q_1(a_1, a_2)+Q_2(a_2, a_4)+Q_3(a_1, a_3)+Q_4(a_3, a_4).$$

Let us begin our optimization with agent 4. To optimize A_4, functions Q_1 and Q_3 are irrelevant. Hence, we obtain:

$$\max_{a_1,a_2,a_3} Q_1(a_1, a_2)+Q_3(a_1, a_3)+\max_{a_4}[Q_2(a_2, a_4)+Q_4(a_3, a_4)].$$

We see that to choose A_4 optimally, the agent must know the values of A_2 and A_3. In effect, it is computing a conditional strategy, with a (possibly) different action choice for each action choice of agents 2 and 3. Agent 4 can summarize the value that it brings to the system in the different circumstances using a new function $f_4(A_2, A_3)$ whose value at the point a_2, a_3 is the value of the internal $\max$ expression. This new function is a new joint value function for agents 2 and 3, summarizing their joint contribution to the total reward under the assumption that agent 4 will act optimally with respect to their choices.

Our problem now reduces to computing

$$\max_{a_1,a_2,a_3} Q_1(a_1, a_2) + Q_3(a_1, a_3) + f_4(a_2, a_3),$$

having one fewer agent. Next, agent 3 makes its decision, giving:

$$\max_{a_1,a_2} Q_1(a_1, a_2) + f_3(a_1, a_2),$$
$$\text{where} \quad f_3(a_1, a_2) = \max_{a_3}[Q_3(a_1, a_3) + f_4(a_2, a_3)].$$

Agent 2 now makes its decision, giving

$$f_2(a_1) = \max_{a_2} Q_1(a_1, a_2) + f_3(a_1, a_2),$$

and agent 1 can now simply choose the action a_1 that maximizes $f_1 = \max_{a_1} f_2(a_1)$. The result at this point is a number, which is the desired maximum over $a_1, \ldots, a_4$.

We can recover the maximizing set of actions by performing the entire process in reverse: The maximizing choice for f_1 selects the action a_1^* for agent 1. To fulfill its commitment to agent 1, agent 2 must choose the value a_2^* which maximizes $f_2(a_1^*)$. This, in turn, forces agent 3 and then agent 4 to select their actions appropriately.

In general, the algorithm maintains a set $\mathcal{F}$ of functions, which initially contains $\{Q_1, \ldots, Q_g\}$. The algorithm then repeats the following steps:

1. Select an uneliminated agent A_l;
2. Take all $f_1, \ldots, f_L \in \mathcal{F}$ whose scope contains A_l.
3. Define a new function $f = \max_{a_l} \sum_j f_j$ and introduce it into $\mathcal{F}$. The scope of f is $\cup_{j=1}^{L} \mathrm{Scope}[f_j] - \{A_l\}$.

As above, the maximizing action choices are recovered by sending messages in the reverse direction. We note that this algorithm is essentially a special case of the algorithm used to solve influence diagrams with multiple parallel decisions [8] (as is the one in the next section). However, to our knowledge, these ideas have not yet been applied in the context of reinforcement learning.

The computational cost of this algorithm is linear in the number of new "function values" introduced in the elimination process. More precisely, consider the computation of a new function e whose scope is $\mathbf{Z}$. To compute this function, we need to compute $|\mathrm{Dom}[\mathbf{Z}]|$ different values. The cost of the algorithm is linear in the overall number of these values, introduced throughout the algorithm. As shown in [6], this cost is exponential in the induced width of the coordination graph for the problem. The algorithm is *distributed* in the sense that the only communication required is between agents that participate in the interior maximizations described above. There is no need for a direct exchange of information between other agents. Thus, the induced tree width has a natural interpretation in this context: It is the maximum number of agents who will need to directly collaborate on the action choice. The order in which the variables are eliminated will have an important impact on the efficiency of this algorithm. We assume that this is determined *a priori* and known to all agents.

At this point, we have shown that if the global utility function Q is approximated by the sum of local utilities Q_j, then it is possible to use the coordination graph to compute the maximizing joint action efficiently. In the remainder of this paper, we will show how we can learn these local utilities efficiently.

3 Markov decision processes

The mechanism described above can used to maximize not just immediate value, but long term cumulative rewards, i.e., the true expected, discounted value of an action, by using Q-functions that are derived from value function from an MDP. The extent to which such a scheme will be successful will be determined by our ability to represent the value function in a form that is usable by our action selection mechanism. Before addressing this question, we first review the MDP framework.

We assume that the underlying control problem is a *Markov Decision Process* (MDP). An MDP is defined as a 4-tuple $(\mathbf{X}, \mathcal{A}, P, R)$ where: $\mathbf{X}$ is a finite set of states; $\mathcal{A}$ is a finite set of actions; P is a *Markovian transition model* where $P(s, a, s')$ represents the probability of going from state s to state s' with action a; and R is a *reward function* $R : \mathbf{X} \times \mathcal{A} \times \mathbf{X} \mapsto \mathbb{R}$, such that $R(s, a, s')$ represents the reward obtained when taking action a in state s and ending up in state s'. For convenience, we will sometimes use $\mathcal{R}(s, a) = \sum_{s'} P(s, a, s')R(s, a, s')$.

We will be assuming that the MDP has an infinite horizon and that future rewards are discounted exponentially with a discount factor $\gamma \in [0, 1)$. A stationary policy π for an MDP is a mapping $\pi : S \mapsto A$, where $\pi(\mathbf{x})$ is the action the agent takes at state $\mathbf{x}$. The optimal value function $\mathcal{V}^*$ is defined so that the value of a state must be the maximal value achievable by any action at that state. More precisely, we define:

$$Q_{\mathcal{V}}(\mathbf{x}, a) = R(\mathbf{x}, a) + \gamma \sum_{\mathbf{x}'} P(\mathbf{x}' \mid \mathbf{x}, a)\mathcal{V}(\mathbf{x}');$$

and the *Bellman operator* $\mathcal{T}^*$ to be:

$$\mathcal{T}^*\mathcal{V}(\mathbf{x}) = \max_a Q_{\mathcal{V}}(\mathbf{x}, a).$$

The optimal value function $\mathcal{V}^*$ is the fixed point of $\mathcal{T}^*$: $\mathcal{V}^* = \mathcal{T}^*\mathcal{V}^*$.

For any value function $\mathcal{V}$, we can define the policy obtained by acting greedily relative to $\mathcal{V}$. In other words, at each state, we take the action that maximizes the one-step utility, assuming that $\mathcal{V}$ represents our long-term utility achieved at the next state. More precisely, we define $\mathit{Greedy}(\mathcal{V})(\mathbf{x}) = \arg\max_a Q_{\mathcal{V}}(\mathbf{x}, a)$. The greedy policy relative to the optimal value function $\mathcal{V}^*$ is the optimal policy $\pi^* = \mathit{Greedy}(\mathcal{V}^*)$.

Recall that we are interested in computing local utilities Q_j for each agent that will represent an approximation to the global utility Q. In [7], we presented an algorithm for computing such value functions for the case where the model is known and represented as a factored MDP. In this paper, we consider the case of unknown action and transition models, i.e., the reinforcement learning case. In the next three sections, we will present alternative, and complementary, approaches for coordinating agents in order to learn the local Q_j functions.

4 Coordination structure in Q-learning

Q-learning is a standard approach to solving an MDP through reinforcement learning. In Q-learning the agent directly learns the values of state-action pairs from observations of quadruples of the form (state, action, reward, next-state), which we will henceforth refer to as $(\mathbf{x}, a, r, \mathbf{x}')$. For each such quadruple, Q-learning performs the following update:

$$Q(\mathbf{x}, a) \leftarrow Q(\mathbf{x}, a) + \alpha[r + \gamma V(\mathbf{x}') - Q(\mathbf{x}, a)],$$

where α, is the "learning rate," or step size parameter and $V(\mathbf{x}') = max_a Q(\mathbf{x}', a)$. With a suitable decay schedule for the learning rate, a policy that ensures that every state-action pair is experienced infinitely often and a representation for $Q(\mathbf{x}, a)$ which can assign an independent value to every state-action pair, Q-learning will converge to estimates for $Q(\mathbf{x}, a)$ which reflect the expected, discounted value of taking action a in state $\mathbf{x}$ and proceeding optimally thereafter.

In practice the formal convergence requirements for Q-learning almost never hold because the state space is too large to permit an independent representation of the value of every state. Typically, a parametric function approximator such as a neural network is used to represent the Q function for each action. The following gradient-based update scheme is then used:

$$\mathbf{w} \leftarrow Q(\mathbf{x}, a, \mathbf{w}) + \alpha[r + \gamma V(\mathbf{x}') - Q(\mathbf{x}, a, \mathbf{w})]\nabla_w Q(\mathbf{x}, a, \mathbf{w}), \quad (1)$$

where $\mathbf{w}$ is a weight vector for our function approximation architecture and, again, the value $V(\mathbf{x}')$ of the next state $\mathbf{x}'$ is given by:

$$V(\mathbf{x}') = max_a Q(\mathbf{x}', a). \quad (2)$$

The Q-learning update mechanism is completely generic and requires only that the approximation architecture is differentiable. We are free to choose an architecture that is compatible with our action selection mechanism. Therefore, we can assume that every agent maintains a local Q_j-function defined over some subset of the state variables, its own actions, and possibly actions of some other agents. The global Q-function is now a function of the state and an action vector $\mathbf{a}$, which contains the joint action of all of the agents defined as the sum of these local Q_j-functions:

$$Q(\mathbf{x}, \mathbf{a}, \mathbf{w}) = \sum_{j=1}^{g} Q_j(\mathbf{x}, \mathbf{a}, \mathbf{w}_j).$$

The are some somewhat subtle consequences of this representation. The first is that determining $V(\mathbf{x}')$ in Eq. (2) seems intractable, because it requires a maximization over an exponentially large action space. Fortunately, the Q-function is factored as a linear combination of local Q_j functions. Thus, we can apply the coordination graph procedure from Section 2 to obtain the maximum value $V(\mathbf{x}')$ at any given state $\mathbf{x}'$.

The Q_j-functions themselves can be maintained locally by each agent as an arbitrary, differentiable function of a set of local weights $\mathbf{w}_j$. Each Q_j can be defined over the entire state space, or just some subset of the state variables visible to agent j.

Once we have defined the local Q_j functions, we must compute the weight update in Eq. (1). Each agent must know:

$$\Delta(\mathbf{x}, a, r, \mathbf{x}', \mathbf{w}) = [r + \gamma V(\mathbf{x}') - Q(\mathbf{x}, a, \mathbf{w})], \quad (3)$$

the difference between the current Q-value and the discounted value of the next state. We have just shown that it is possible to apply the coordination graph from Section 2 to compute $V(\mathbf{x}')$. The other unknown terms in Eq. (3) are the reward r and the previous Q value, $Q(\mathbf{x}, a, \mathbf{w})$. The reward is observed and the Q value can be computed by a simple message passing scheme similar to the one used in the coordination graph by fixing the action of every agent to the one assigned in a.

Therefore, after the coordination step, each agent will have access to the value of $\Delta(\mathbf{x}, a, r, \mathbf{x}', \mathbf{w})$. At this point, the weight update equation is entirely local:

$$\mathbf{w}_i \leftarrow Q(\mathbf{x}, a, \mathbf{w}) + \alpha\ \Delta(\mathbf{x}, a, r, \mathbf{x}', \mathbf{w})\nabla_{w_i} Q_i(\mathbf{x}, a, \mathbf{w}_i),$$

The reason is simply that the gradient decomposes linearly: Once an action is selected, there are no cross-terms involving any $\mathbf{w}_i$ and $\mathbf{w}_j$. The locality of the weight updates in this formulation of Q-learning make it very attractive for a distributed implementation. Each agent can maintain an entirely local Q-function and does need to know anything about the structure of the neighboring agents' Q-functions. Different agents can even use different architectures, e.g., one might use a neural network and another might use a CMAC. The only requirement is that the joint Q-function be expressed as a sum of the these individual Q-functions.

The only negative aspect of this Q-learning formulation is that, like almost all forms of Q-learning with function approximation, it is difficult to provide any kind of formal convergence guarantees. We have no reason to believe that it would perform any better or worse than general Q-learning with a linear function approximation architecture, however this remains to be verified experimentally.

5 Multiagent LSPI

Least Squares policy iteration (LSPI) [11] is a new reinforcement learning method that performs policy iteration by using a stored corpus of samples in place of a model. LSPI represents Q-functions using as a linear combination of basis functions. Given a policy, π_i, LSPI computes a set of Q-functions, Q_{π_i} (in the space spanned by the basis functions) which are a fixed point for π with respect to the samples. The new Q_{π_i} then implicitly define policy π_{i+1} and the process is repeated until some form of convergence is achieved.

We briefly review the mathematical operations required for LSPI here; full details are available in [10]. We assume that our Q-functions will be approximated by a linear combination of basis functions (features),

$$\widehat{Q}^{\pi}(\mathbf{x}, a, w) = \sum_{i=1}^{k} \phi_i(\mathbf{x}, a) w_i = \phi(\mathbf{x}, a)^{\intercal} w,$$

For convenience we express our basis functions in matrix form:

$$\mathbf{\Phi} = \begin{pmatrix} \phi(\mathbf{x}_1, a_1)^{\intercal} \\ \cdots \\ \phi(\mathbf{x}, a)^{\intercal} \\ \cdots \\ \phi(\mathbf{x}_{|\mathbf{X}|}, a_{|A|})^{\intercal} \end{pmatrix}.$$

where Φ is $(|\mathbf{X}||A| \times k)$. If we knew the transition matrix, P^{π}, for the current policy and knew the reward function we could, in principle, compute the fixed point by defining and solving the following system:

$$\mathbf{A} w^{\pi} = b,$$

where

$$\mathbf{A} = \mathbf{\Phi}^{\intercal}(\mathbf{\Phi} - \gamma \mathbf{P}^{\pi}\mathbf{\Phi})$$

and

$$b = \mathbf{\Phi}^{\intercal} R.$$

In practice, we must sample experiences with the environment in place of $\mathcal{R}$ and P^{π}. Given a set of samples, $D = \{s_{d_i}, a_{d_i}, s'_{d_i}, r_{d_i})|\ i = 1, 2, \ldots, L\}$, where the (s_{d_i}, a_{d_i}), we can construct approximate versions of $\mathbf{\Phi}$, $\mathbf{P}^{\pi}\mathbf{\Phi}$, and $\mathcal{R}$ as follows :

$$\widehat{\mathbf{\Phi}} = \begin{pmatrix} \phi(s_{d_1}, a_{d_1})^\intercal \\ \cdots \\ \phi(s_{d_i}, a_{d_i})^\intercal \\ \cdots \\ \phi(s_{d_L}, a_{d_L})^\intercal \end{pmatrix} \quad \widehat{\mathbf{P}^\pi \mathbf{\Phi}} = \begin{pmatrix} \phi(s'_{d_1}, \pi\left(s'_{d_1}\right))^\intercal \\ \cdots \\ \phi(s'_{d_i}, \pi\left(s'_{d_i}\right))^\intercal \\ \cdots \\ \phi(s'_{d_L}, \pi\left(s'_{d_L}\right))^\intercal \end{pmatrix}$$

$$\widehat{\mathcal{R}} = \begin{pmatrix} r_{d_1} \\ \cdots \\ r_{d_i} \\ \cdots \\ r_{d_L} \end{pmatrix}$$

It's easy to see that approximations $(\widehat{\mathbf{A}}_1, \widehat{b}_1, \widehat{\mathbf{A}}_2, \widehat{b}_2)$ derived from different sets of samples (D_1, D_2) can be combined additively to yield a better approximation that corresponds to the combined set of samples:

$$\widehat{\mathbf{A}} = \widehat{\mathbf{A}}_1 + \widehat{\mathbf{A}}_2 \qquad \text{and} \qquad \widehat{b} = \widehat{b}_1 + \widehat{b}_2.$$

This observation leads to an incremental update rule for $\widehat{\mathbf{A}}$ and $\widehat{b}$. Assume that initially $\widehat{\mathbf{A}} = 0$ and $\widehat{b} = 0$. For a fixed policy, a new sample $(\mathbf{x}, a, r, \mathbf{x}')$ contributes to the approximation according to the following update equation :

$$\widehat{\mathbf{A}} \leftarrow \widehat{\mathbf{A}} + \phi(\mathbf{x}, a)\Big(\phi(\mathbf{x}, a) - \gamma\phi(\mathbf{x}', \pi(\mathbf{x}'))\Big)^\intercal$$

and

$$\widehat{b} \leftarrow \widehat{b} + \phi(\mathbf{x}, a)r$$

We note that this approach is very similar to the LSTD algorithm [5]. Unlike LSTD, which defines a system of equations relating state values to state values, LSPI is defined over Q-values. Each iteration of LSPI yields the Q-values for the current policy. Thus, each solution implicitly defines the next policy for policy iteration.

An important feature of LSPI is that it is able to reuse the same set of samples even as the policy changes. For example, suppose the corpus contains a transition from state $\mathbf{x}$ to state $\mathbf{x}'$ under action a_1 and $\pi_i(\mathbf{x}') = a_2$. This is entered into the $\mathbf{A}$ matrix as if a transition were made from $Q(\mathbf{x}, a_1)$ to $Q(\mathbf{x}', a_2)$. If $\pi_{i+1}(\mathbf{x}')$ changes the action for $\mathbf{x}'$ from a_2 to a_3, then the next iteration of LSPI enters a transition from $Q(\mathbf{x}, a_1)$ to $Q(\mathbf{x}', a_3)$ into the $\mathbf{A}$ matrix. The sample can be reused because the dynamics for state $\mathbf{x}$ under action a_1 have not changed.

The application of collaborative action selection to the LSPI framework is surprisingly straightforward. We first note that since LSPI is a linear method, any set of Q-functions produced by LSPI will, by construction, be of the right form for collaborative action selection. Each agent is assigned a local set of basis functions which define its local Q-function. These basis functions can be defined over the agent's own actions as well as the actions of a small number of other agents. As with ordinary LSPI, the current policy π_i is defined implicitly by the current set of Q-functions, Q^{π_i}. However, in the multiagent case, we cannot enumerate each possible action to determine the policy at some given state because this set of actions is exponential in the number of agents. Fortunately, we can again exploit the structure of the coordination graph to determine the optimal actions relative to Q^{π_i}: For each transition from state $\mathbf{x}$ to state $\mathbf{x}'$ under joint action $\mathbf{a}$ the coordination graph is used to determine the optimal actions for $\mathbf{x}'$, yielding action $\mathbf{a}'$. The transition is added to the A matrix as a transition from $Q(\mathbf{x}, \mathbf{a})$ to $Q(\mathbf{x}', \mathbf{a}')$.

An advantage to the LSPI approach to collaborative action selection is that it computes a value function for each successive policy which has a coherent interpretation as a projection into the linear space spanned by the individual agent's local Q-functions. Thus, there is reason to believe that the resulting Q-functions will be well-suited to collaborative action selection using a coordination graph mechanism.

A disadvantage of the LSPI approach is that it is not currently amenable to a distributed implementation during the learning phase: Construction of the $\mathbf{A}$ matrix requires knowledge of the evaluation of each agent's basis functions for every state in the corpus, not only for every action that is actually taken, but for every action recommended by every policy considered by LSPI.

6 Coordination in direct policy search

Value function based reinforcement learning methods have recently come under some criticism as being unstable and difficult to use in practice. A function approximation architecture that is not well-suited to the problem can diverge or produce poor results with little meaningful feedback that is directly useful for modifying the function approximator to achieve better performance.

LSPI was designed to address some of the concerns with Q-learning based value function approximation. It is more stable than Q-learning and since it is a linear method, it is somewhat easier to debug. However, LSPI is still an approximate policy iteration procedure and can be quite sensitive to small errors in the estimated Q-values for policies [4]. In practice, LSPI can to take large, coarse steps in policy space.

The shortcomings of value function based methods have led to a surge of interest in direct policy search methods [13; 12; 1; 15; 9]. These methods use gradient ascent to search a space of parameterized stochastic policies. As with all gradient ascent methods, local optima can be problematic. Defining a relatively smooth but expressive policy space and finding reasonable starting points within this space are all important elements of any successful application of gradient ascent.

We now show how to seed a gradient ascent procedure with a multiagent policy generated by Q-learning or LSPI as described above. To guarantee that the gradient exists, policy search methods require stochastic policies. Our first task is to convert the deterministic policy implied by our value Q-functions into a stochastic, policy $\mu(\mathbf{a}|\mathbf{x})$, i.e., a distribution over actions given the state. A natural way to do this, which also turns out to be compatible with most policy search methods, is to create a softmax over the Q-values:

$$\mu(\mathbf{a} \mid \mathbf{x}) = \frac{e^{\sum_j Q_j(\mathbf{x}, \mathbf{a})}}{\sum_b e^{\sum_k Q_k(\mathbf{x}, \mathbf{b})}}. \tag{4}$$

To be able to apply policy search methods for such policy representation, we must present two additional steps: The first is a method of efficiently generating samples from our stochastic policy and the second is a method of efficiently differentiating our stochastic policy for gradient ascent purposes.

Sampling from our stochastic policy may appear problematic because of the size of the joint action space. For sampling purposes, we can ignore the denominator, since it is the same for all actions, and sample from the numerator directly as an unnormalized potential function. To do this sampling we again use variable elimination on a coordination graph with exactly the same structure as the one in Section 2. Conditioning on the current state $\mathbf{x}$ is again easy: each agent needs to observe only the variables in $Observable[Q_j]$ and instantiate Q_j appropriately. At this point, we need to generate a sample from Q_j functions that depend only on the action choice.

Following our earlier example, our task is now to sample from the potential corresponding to the numerator of $\mu(\mathbf{a} \mid \mathbf{x})$. Suppose, for example, that the individual agent's Q-functions have the following form:

$$V = Q_1(a_1, a_2) + Q_2(a_2, a_4) + Q_3(a_1, a_3) + Q_4(a_3, a_4).$$

and we wish to sample from the potential function for

$$e^{Q_1(a_1,a_2)} e^{Q_2(a_2,a_4)} e^{Q_3(a_1,a_3)} e^{Q_4(a_3,a_4)}.$$

To sample actions one at a time, we will follow a strategy of marginalizing out actions until we are left with a potential over a single action. We then sample from this potential and propagate the results backwards to sample actions for the remaining agents. Suppose we begin by eliminating a_4. Agent 4 can summarize it's impact on the rest of the distribution by combining its potential function with that of agent 2 and defining a new potential:

$$f_4(A_2, A_3) = \sum_{A_4} e^{Q_2(a_2,a_4)} e^{Q_4(a_3,a_4)}.$$

The problem now reduces to sampling from

$$e^{Q_1(a_1,a_2)} e^{Q_3(a_1,a_3)} f_4(a_2, a_3),$$

having one fewer agent. Next, agent 3 communicates its contribution giving:

$$e^{Q_1(a_1,a_2)} f_3(a_1, a_2),$$
$$\text{where} \quad f_3(a_1, a_2) = \textstyle\sum_{a_3} e^{Q_3(a_1,a_3)} f_4(a_2, a_3).$$

Agent 2 now communicates its contribution, giving

$$f_2(a_1) = \sum_{a_2} e^{Q_1(a_1,a_2)} f_3(a_1, a_2),$$

and agent 1 can now sample actions from the potential $P(a_1) \sim f_2(a_1)$.

We can now sample actions for the remaining agents by reversing the direction of the messages and sampling from the distribution for each agent, conditioned on the choices of the previous agents. For example, when agent 2 is informed of the action selected by agent 1, agent 2 can sample actions from the distribution:

$$P(a_2|a_1) = \frac{P(a_2, a1)}{P(a1)} = \frac{e^{Q_1(a_1,a_2)} f_3(a_1, a_2)}{f_2(a_1)}.$$

The general algorithm has the same message passing topology as the original action selection mechanism. The only difference is the content of the messages: The forward pass messages are probability potentials and the backward pass messages are used to compute conditional distributions from which actions are sampled.

The next key operation is the computation of the gradient of our stochastic policy function, a key operation in a REINFORCE style [16] policy search algorithm.[1] First, recall that we are using a linear representation for each local Q_j function:

$$Q_j(\mathbf{x}, \mathbf{a}) = \sum_i w_{i,j} \phi_{i,j}(\mathbf{x}, \mathbf{a});$$

where each basis function $\phi_{i,j}(\mathbf{x}, \mathbf{a})$ can depend on a subset of the state and action variables. Our stochastic policy representation now becomes:

$$\mu(\mathbf{a} \mid \mathbf{x}) = \frac{e^{\sum_{j,i} w_{i,j} \phi_{i,j}(\mathbf{x},\mathbf{a})}}{\sum_b e^{\sum_{k,i} w_{i,k} \phi_{i,k}(\mathbf{x},\mathbf{a})}}.$$

To simplify our notation, we will refer to the weights as w_i rather than $w_{i,j}$ and the basis functions as ϕ_i rather than $\phi_{i,j}$. Finally, we need to compute the gradient of the log policy:

$$\begin{aligned}
&\frac{\partial}{\partial w_i} \ln \mu(\mathbf{a}|\mathbf{x}) = \\
&= \frac{\partial}{\partial w_i} \ln \left(\frac{e^{\sum_i w_i \phi_i(\mathbf{x},\mathbf{a})}}{\sum_b e^{\sum_i w_i \phi_i(\mathbf{x},\mathbf{b})}} \right); \\
&= \frac{\partial}{\partial w_i} \ln \left(e^{\sum_i w_i \phi_i(\mathbf{x},\mathbf{a})} \right) - \\
&\quad \frac{\partial}{\partial w_i} \ln \left(\sum_b e^{\sum_i w_i \phi_i(\mathbf{x},\mathbf{b})} \right); \\
&= \phi_i(\mathbf{x}, \mathbf{a}) - \frac{\frac{\partial}{\partial w_i} \sum_b e^{\sum_i w_i \phi_i(\mathbf{x},\mathbf{b})}}{\sum_{b'} e^{\sum_i w_i \phi_i(\mathbf{x},\mathbf{b}')}}; \\
&= \phi_i(\mathbf{x}, \mathbf{a}) - \frac{\sum_b \frac{\partial}{\partial w_i} e^{\sum_i w_i \phi_i(\mathbf{x},\mathbf{b})}}{\sum_{b'} e^{\sum_i w_i \phi_i(\mathbf{x},\mathbf{b}')}}; \\
&= \phi_i(\mathbf{x}, \mathbf{a}) - \sum_b \phi_i(\mathbf{x}, \mathbf{b}) \frac{e^{\sum_i w_i \phi_i(\mathbf{x},\mathbf{b})}}{\sum_{b'} e^{\sum_i w_i \phi_i(\mathbf{x},\mathbf{b}')}}.
\end{aligned}$$

We note that both the numerator and the denominator in the summation can be determined by a variable elimination procedure similar to the stochastic action selection procedure.

These action selection and gradient computation mechanisms provide the basic functions required for essentially any policy search method. As in the case of Q-learning, a global error signal must be shared by the entire set of agents. Apart from this, the gradient computations and stochastic policy sampling procedure involve a message passing scheme with the same topology as the action selection mechanism. We believe that these methods can be incorporated into any of a number of policy search methods to fine tune a policy derived by Q-learning with linear Q-functions or by LSPI.

7 Experimental results

In this section we report results of applying our Coordinated RL approach with LSPI to the SysAdmin multiagent domain [7]. The SysAdmin problem consists of a network of n machines connected in one of the following topologies: chain, ring, star, ring-of-rings, or star-and-ring. The state of each machine j is described by two variables: status $S_j \in \{\text{good}, \text{faulty}, \text{dead}\}$, and load $L_j \in$

[1] Most policy search algorithms are of this style.

{idle, loaded, process successful}. New jobs arrive with probability 0.5 on idle machines and make them loaded. A loaded machine in good status can execute and successfully complete a job with probability 0.5 at each time step. A faulty machine can also execute jobs, but it will take longer to terminate (jobs end with probability 0.25). A dead machine is not able to execute jobs and remains dead until it is rebooted. Each machine receives a reward of $+1$ for each job completed successfully, otherwise it receives a reward of 0. Machines fail stochastically and switch status from good to faulty and eventually to dead, but they are also influenced by their neighbors; a dead machine increases the probability that its neighbors will become faulty and eventually die.

Each machine is also associated with an agent A_j which at each step has to choose between rebooting the machine or not. Rebooting a machine makes its status good independently of the current status, but any running job is lost. These agents have to coordinate their actions so as to maximize the total reward for the system, or in other words, maximize the total number of successfully completed jobs in the system. Initially, all machines are in "good" status and the load is set to "process successful". The discount factor γ is set to 0.95.

The SysAdmin problem has been studied in [7], where the model of the process is assumed to be available as a factored MDP. This is a fairly large MDP with 9^n possible states and 2^n possible actions for a network of n machines. A direct approach to solving such an MDP will fail even for small values of n. The state value function is approximated as a linear combination of indicator basis functions, and the coefficients are computed using a Linear Programming (LP) approach. The derived policies are very close to the theoretical optimal and significantly better compared to policies learned by the Distributed Reward (DR) and Distributed Value Function (DVF) algorithms [14].

We use the SysAdmin problem to evaluate our coordinated RL approach despite the availability of a factored model because it provides a baseline for comparing the model-based approach with coordinated RL. Note that conventional RL methods would fail here because of the exponential action space. We applied Coordinated RL approach with LSPI in these experiments. To make a fair comparison with the results obtained by the LP approach, we had to make sure that the sets of basis functions used in each case are comparable. The LP approach makes use of "single" or "pair" indicator functions to approximate the state value function $V(\mathbf{x})$ and forms Q values using the model and the Bellman equation. It then solves a linear program to find the coefficients for the approximation of $V(\mathbf{x})$ under a set of constraints imposed by the Q values.

On the other hand, LSPI uses a set of basis functions to approximate the state-action value function $Q(\mathbf{x}, a)$. To make sure the two sets are comparable, we have to choose basis functions for LSPI that span the space of all possible Q functions that the LP approach can form. To do this we backprojected the LP method's basis functions through the transition model[7], added the instantaneous reward, and used the resulting functions as our basis for LSPI. We note that this set of basis functions potentially spans a larger space than the Q functions considered by the LP approach. This is because the LP constrains its Q functions by a lower-dimensional V function, while LSPI is free to consider the entire space spanned by its Q-function basis. While there are some slight differences, we believe bases are close enough to permit a fair comparison.

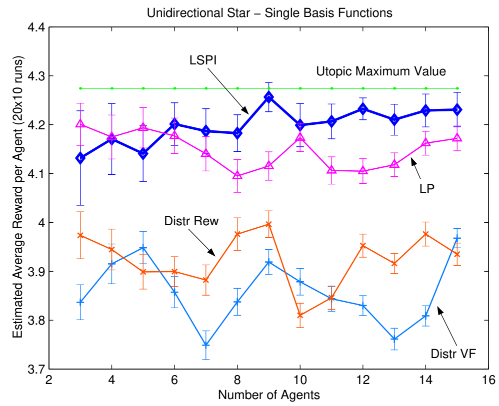

Figure 2: Results on star networks.

In our experiments, we created two sets of LSPI basis functions corresponding to the backprojections of the "single" or "pair" indicator functions from [7]. The first set of experiments involved a star topology with up to 14 legs yielding a range from 3 to 15 agents. For n machines, we experimentally found that about $600n$ samples are sufficient for LSPI to learn a good policy. The samples were collected by starting at the initial state and following a purely random policy. To avoid biasing our samples too heavily by the stationary distribution of the random policy, each episode was truncated at 15 steps. Thus, samples were collected from $40n$ episodes each one 15 steps long. The resulting policies were evaluated with a Monte-Carlo approach by averaging 20 100-step long runs.

The entire experiment was repeated 10 times with different sample sets and the results were averaged. Figure 2 shows the results obtained by LSPI compared with the results of LP, DR, and DVF as reported in [7] for the "single" basis functions case. The error bars for LSPI correspond to the averages over the different sample sets used. The "pair" basis functions produced similar results for LSPI and slightly better for LP and are not reported here due to space limitations.

The second set of experiments involved a ring-of-rings topology with up to 6 small rings (each one of size 3) forming a bigger ring and the total number of agents was ranging from 6 to 18. All learning parameters were the same as in the star case. Figure 3 compares the results of LSPI with those of LP, DR, and DVF for the "single" basis functions case.

The results in both cases clearly indicate that LSPI learns very good policies comparable to the LP approach using the same basis functions, but *without* any use of the model. It is worth noting that the number of samples used in each case grows linearly in the number of agents, whereas the joint state-action space grows exponentially.

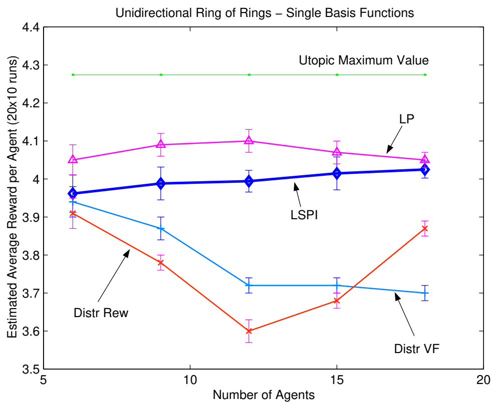

Figure 3: Results on ring-of-rings networks.

8 Conclusions and future work

In this paper, we proposed a new approach to reinforcement learning: *Coordinated RL*. In this approach, agents make coordinated decisions and share information to achieve a principled learning strategy. Our method successfully incorporates the cooperative action selection mechanism derived in [7] into the reinforcement learning framework to allow for structured communication between agents, each of which have only partial access to the state description. We presented three instances of Coordinated RL, extending three RL frameworks: Q-learning, LSPI and policy search. With Q-learning and policy search, the learning mechanism can be distributed. Agents communicate reinforcement signals, utility values and conditional policies. On the other hand, in LSPI some centralized coordination is required to compute the projection of the value function. In all cases, the resulting policies can be executed in a distributed manner. In our view, an algorithm such as LSPI can provide a batch offline estimate of the Q-functions. Subsequently, Q-learning or direct policy search can be applied online to refine this estimate. By using our Coordinated RL method, we can smoothly shift between these two phases.

Our initial empirical results demonstrate that reinforcement learning methods can learn good policies using the type of linear architecture required for cooperative action selection. In our experiments with LSPI, we reliably learned policies that were comparable to the best policies achieved by other methods and close to the theoretical optimal achievable in our test domains. The amount of data required to learn good policies scaled linearly with the number of state and action variables even though the state and action spaces were growing exponentially.

Our initial experiments involved models with discrete state and action spaces that could be described compactly using a dynamic Bayesian network. These were chosen primarily to compare learning performance with previous closed-form approximation methods and our basis functions were chosen to match closely the basis functions used in earlier experiments with the same models. In future work, we plan to explore problems that involve continuous variables and basis functions. While our methods do require discrete actions, they generalize immediately to continuous state variables.

Guestrin et al. [7] presented the coordination graph method for action selection used in this paper and an algorithm for finding an approximation to the Q-function using local Q_j functions for problems where the MDP model can be represented compactly as a factored MDP. The Multiagent LSPI algorithm presented in this paper can be thought of as a *model-free* reinforcement learning approach for generating the local Q_j approximation. The factored MDP approach [7] is able to exploit the model and compute its approximation very efficiently. On the other hand, Multiagent LSPI is more general as it can be applied to problems that cannot be represented compactly by a factored MDP.

Acknowledgments We are very grateful to Daphne Koller for many useful discussions. This work was supported by the ONR under the MURI program "Decision Making Under Uncertainty" and by the Sloan Foundation. The first author was also supported by a Siebel Scholarship.

References

[1] J. Baxter and P.Bartlett. Reinforcement learning in POMDP's via direct gradient ascent. In *Proc. Int. Conf. on Machine Learning (ICML)*, 2000.

[2] D. Bernstein, S. Zilberstein, and N. Immerman. The complexity of decentralized control of Markov decision processes. In *Proc. of Conf. on Uncertainty in AI (UAI-00)*, 2000.

[3] U. Bertele and F. Brioschi. *Nonserial Dynamic Programming*. Academic Press, New York, 1972.

[4] D. Bertsekas and J. Tsitsiklis. *Neuro-Dynamic Programming*. Athena Scientific, Belmont, Massachusetts, 1996.

[5] S. Bradtke and A. Barto. Linear least-squares algorithms for temporal difference learning. *Machine Learning*, 2(1):33–58, January 1996.

[6] R. Dechter. Bucket elimination: A unifying framework for reasoning. *Artificial Intelligence*, 113(1–2):41–85, 1999.

[7] Carlos Guestrin, Daphne Koller, and Ronald Parr. Multiagent planning with factored MDPs. In *14th Neural Information Processing Systems (NIPS-14)*, 2001.

[8] F. Jensen, F. V. Jensen, and S. Dittmer. From influence diagrams to junction trees. In *UAI-94*, 1994.

[9] V. Konda and J. Tsitsiklis. Actor-critic algorithms. In *NIPS-12*, 2000.

[10] M. G. Lagoudakis and R. Parr. Model-Free Least-Squares policy iteration. Technical Report CS-2001-05, Department of Computer Science, Duke University, December 2001.

[11] M. Lagoudakis and R. Parr. Model free least squares policy iteration. In *NIPS-14*, 2001.

[12] A. Ng and M. Jordan. PEGASUS: A policy search method for large MDPs and POMDPs. In *UAI-00*, 2000.

[13] A. Ng, R. Parr, and D. Koller. Policy search via density estimation. In *NIPS-12*, 2000.

[14] J. Schneider, W. Wong, A. Moore, and M. Riedmiller. Distributed value functions. In *16th ICML*, 1999.

[15] R. Sutton, D. McAllester, S. Singh, and Y. Mansour. Policy gradient methods for reinforcement learning with function approximation. In *NIPS-12*, 2000.

[16] Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8(3):229–256, 1992.

[17] D. Wolpert, K. Wheller, and K. Tumer. General principles of learning-based multi-agent systems. In *Proc. of 3rd Int. Conf. on Autonomous Agents (Agents'99)*, 1999.