Least-Squares Methods in Reinforcement Learning for Control

Michail G. Lagoudakis[1], Ronald Parr[1], and Michael L. Littman[2]

[1] Department of Computer Science, Duke University, Durham, NC 27708, U.S.A.
{mgl,parr}@cs.duke.edu

[2] Shannon Laboratory, AT&T Labs – Research, Florham Park, NJ 07932, U.S.A.
mlittman@research.att.com

Abstract. Least-squares methods have been successfully used for prediction problems in the context of reinforcement learning, but little has been done in extending these methods to control problems. This paper presents an overview of our research efforts in using least-squares techniques for control. In our early attempts, we considered a direct extension of the Least-Squares Temporal Difference (LSTD) algorithm in the spirit of Q-learning. Later, an effort to remedy some limitations of this algorithm (approximation bias, poor sample utilization) led to the Least-Squares Policy Iteration (LSPI) algorithm, which is a form of model-free approximate policy iteration and makes efficient use of training samples collected in any arbitrary manner. The algorithms are demonstrated on a variety of learning domains, including algorithm selection, inverted pendulum balancing, bicycle balancing and riding, multiagent learning in factored domains, and, recently, on two-player zero-sum Markov games and the game of Tetris.

1 Introduction

Linear least-squares methods have been successfully used for prediction problems in the context of reinforcement learning. Although these methods lack the generalization ability of “black box” methods such as neural networks, they are much easier to implement and debug. It is also easier to understand why a linear method succeeds or fails, to quantify the importance of each basis feature, and to engineer these features for better performance. For example, the *Least Squares Temporal Difference* learning algorithm (LSTD) [2] makes efficient use of data and converges faster than conventional temporal difference learning methods.

Unfortunately, little has been done in extending these methods to control problems. Using LSTD directly as part of a policy iteration algorithm can be problematic, as was shown by Koller and Parr [6]. This failure is partly due to the fact that LSTD approximations are biased by the stationary distribution of the underlying Markov chain. However, even if this problem is solved, the state value function that LSTD learns is of no use for policy improvement since a model of the process is not available, in general, for learning control problems.

This paper is an overview of our research efforts in using least-squares techniques for learning control problems. First, we consider *Least-Squares Q-learning* (LSQL), an extension of LSTD that learns a state-action value function (instead of a state value function) in the spirit of Q-learning. Then, we present the *Least-Squares Policy Iteration* (LSPI) algorithm which is a form of model-free approximate policy iteration and resolves some limitations of LSQL (approximation bias, poor sample utilization). The algorithms were tested and produced excellent results on a variety of learning domains, including algorithm selection, inverted pendulum balancing, bicycle balancing and riding, and multiagent learning in factored domains. Currently, LSPI is being tested on the game of Tetris and on two-player zero-sum Markov games.

2 MDPs and Reinforcement Learning

We assume that the underlying control problem is a *Markov Decision Process* (MDP). An MDP is defined as a 4-tuple $(\mathcal{S}, \mathcal{A}, P, R)$, where: $\mathcal{S} = \{s_1, s_2, ..., s_n\}$ is a finite set of states; $\mathcal{A} = \{a_1, a_2, ..., a_m\}$ is a finite set of actions; P is a Markovian state transition model — $P(s, a, s')$ is the probability of making a transition to state s' when taking action a in state s ($s \xrightarrow{a} s'$); and, R is a reward (or cost) function — $R(s, a, s')$ is the reward for the transition $s \xrightarrow{a} s'$.

We assume that the MDP has an infinite horizon and that future rewards are discounted exponentially with a discount factor $\gamma \in [0, 1)$. Assuming that all policies are proper, i.e. that all episodes eventually terminate, our results generalize to the undiscounted case as well.

A *deterministic policy* π for an MDP is a mapping $\pi : \mathcal{S} \mapsto \mathcal{A}$, where $\pi(s)$ is the action the agent takes at state s. The *state-action value function* $Q^\pi(s, a)$, defined over all possible combinations of states and actions, indicates the expected, discounted, total reward when taking action a in state s and following policy π thereafter. The exact Q-values for all state-action pairs can be found by solving the linear system of the Bellman equations :

$$Q^\pi(s,a) = \mathcal{R}(s,a) + \gamma \sum_{s'} P(s,a,s') Q^\pi(s', \pi(s')) \qquad \text{or} \qquad Q^\pi = \mathcal{R} + \gamma \mathbf{P}^\pi Q^\pi \ ,$$

where Q^π and $\mathcal{R}$ are vectors of size $|\mathcal{S}||\mathcal{A}|$ and $\mathbf{P}^\pi$ is a stochastic matrix of size $(|\mathcal{S}||\mathcal{A}| \times |\mathcal{S}||\mathcal{A}|)$. $\mathcal{R}$ is the expected reward for state-action pairs, $\mathcal{R}(s,a) = \sum_{s'} P(s,a,s')R(s,a,s')$, and $\mathbf{P}^\pi$ describes the probability of transitions from pairs (s,a) to pairs $(s', \pi(s'))$.

For every MDP, there exists a deterministic *optimal policy*, π^*, not necessarily unique, which maximizes the expected, discounted return of every state. The state-action value function Q^{π^*} of an optimal policy is the fixed point of the non-linear Bellman optimality equations:

$$Q^{\pi^*}(s,a) = \mathcal{R}(s,a) + \gamma \max_{a'} \sum_{s'} P(s,a,s') Q^{\pi^*}(s', a') \ .$$

Value Iteration is a method of approximating the Q^{π^*} values arbitrarily closely by iterating the equations above (similar to the Gauss iteration for linear systems). If Q^{π^*} is known, the optimal policy can be constructed simply by finding the maximizing action in each state, $\pi^*(s) = \max_a Q^{\pi^*}(s,a)$. *Policy Iteration* is another method of discovering an optimal policy by iterating through a sequence of monotonically improving policies. Each iteration consists of two phases: *Value Determination* computes the value function for a policy $\pi^{(t)}$ by solving the linear Bellman equations, and *Policy Improvement* defines the next policy as $\pi^{(t+1)}(s) = \arg\max_a Q^{\pi^{(t)}}(s,a)$. These steps are repeated until convergence to an optimal policy, often in a surprisingly small number of steps.

In the absence of a model of the MDP, that is, when P and $\mathcal{R}$ are unknown, the decision maker has to learn the optimal policy through interaction with the environment. Knowledge comes in the form of samples (s,a,r,s'), where s is a state of the process, a is the action taken in s, r is the reward received, and s' is the resulting state. Samples can be collected from actual (sequential) episodes or from queries to a generative model of the MDP. In the extreme case, they can be experiences of other agents on the same MDP. The class of problems that fall under this framework is known as *Reinforcement Learning* (RL) [5,15,1].

Q-learning [17] is a popular algorithm that stochastically approximates Q^{π^*}. It starts with any arbitrary initial guess $\widehat{Q}^{(0)}$ for for the values of Q^{π^*}. For each sample (s,a,r,s') considered, Q-learning makes the update

$$\widehat{Q}^{(t+1)}(s,a) = (1-\alpha)\widehat{Q}^{(t)}(s,a) + \alpha\left[r + \max_{a'}\left\{\widehat{Q}^{(t)}(s',a')\right\}\right] ,$$

where $\alpha \in (0,1]$ is the learning rate. Under certain conditions (e.g., infinitely many samples for each state-action pair, appropriately decreasing learning rate), $\widehat{Q}$ is guaranteed to converge to Q^{π^*}.

3 Least-Squares Methods in Reinforcement Learning

3.1 Least-Squares Approximation of Q Functions

Q functions can be stored in tables of size $|\mathcal{S}||\mathcal{A}|$ for small MDPs. This is, however, impractical for large state and action spaces. In such cases, it is common to approximate Q^π with a parametric function approximator by setting the parameters to a set of values that maximizes the accuracy of the approximator. A common class of approximators, known as *linear architectures*, approximate a value function as a linear combination of k basis functions (features):

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

where w is a set of weights (parameters), and, in general, $k << |\mathcal{S}||\mathcal{A}|$. Let $\boldsymbol{\Phi}$ be the $(|\mathcal{S}||\mathcal{A}| \times k)$ matrix, where row i is the vector $\phi_i(s,a)^\intercal$. We are interested in finding a set of weights w^π that yields a fixed point in value function space, that is, a value function $\widehat{Q}^\pi = \boldsymbol{\Phi} w^\pi$ that is invariant under one step of value determination followed by orthogonal projection to the space spanned by the basis functions. In particular, under the assumption that the columns of $\boldsymbol{\Phi}$ are independent, we require that

$$\boldsymbol{\Phi}(\boldsymbol{\Phi}^\intercal\boldsymbol{\Phi})^{-1}\boldsymbol{\Phi}^\intercal(\mathcal{R} + \gamma\mathbf{P}^\pi\boldsymbol{\Phi} w^\pi) = \boldsymbol{\Phi} w^\pi \implies \boldsymbol{\Phi}^\intercal(\boldsymbol{\Phi} - \gamma\mathbf{P}^\pi\boldsymbol{\Phi})w^\pi = \boldsymbol{\Phi}^\intercal\mathcal{R} \implies \mathbf{A}w^\pi = b ,$$

where $\mathbf{A} = \boldsymbol{\Phi}^\intercal(\boldsymbol{\Phi} - \gamma \mathbf{P}^\pi \boldsymbol{\Phi})$ is a square matrix of size $k \times k$, and $b = \boldsymbol{\Phi}^\intercal \mathcal{R}$. The solution of the system, $w^\pi = \mathbf{A}^{-1} b$, yields the desired set of weights. We note that this is also the standard fixed point approximation method used in the LSTD algorithm with the exception that the problem here is formulated in terms of Q values instead of state values. For any $\mathbf{P}^\pi$, a unique solution is guaranteed to exist for all but finitely many values of γ [6].

3.2 LSQ: Learning the State-Action Value Function

When the model $(\mathcal{R}, \mathbf{P}^\pi)$ of the underlying MDP is not available, $\mathbf{A}$ and b cannot be determined *a priori*, but they can be approximated using samples. Recall that $\boldsymbol{\Phi}$, $\mathbf{P}^\pi\boldsymbol{\Phi}$, and $\mathcal{R}$ are of the form

$$\boldsymbol{\Phi} = \begin{pmatrix} \phi(s_1, a_1)^\intercal \\ \cdots \\ \phi(s, a)^\intercal \\ \cdots \\ \phi(s_{|\mathcal{S}|}, a_{|\mathcal{A}|})^\intercal \end{pmatrix} \quad \mathbf{P}^\pi \boldsymbol{\Phi} = \begin{pmatrix} \sum_{s'} P(s_1, a_1, s')\phi(s', \pi(s'))^\intercal \\ \cdots \\ \sum_{s'} P(s, a, s')\phi(s', \pi(s'))^\intercal \\ \cdots \\ \sum_{s'} P(s_{|\mathcal{S}|}, a_{|\mathcal{A}|}, s')\phi(s', \pi(s'))^\intercal \end{pmatrix} \quad \mathcal{R} = \begin{pmatrix} \sum_{s'} P(s_1, a_1, s')R(s_1, a_1, s') \\ \cdots \\ \sum_{s'} P(s, a, s')R(s, a, s') \\ \cdots \\ \sum_{s'} P(s_{|\mathcal{S}|}, a_{|\mathcal{A}|}, s')R(s_{|\mathcal{S}|}, a_{|\mathcal{A}|}, s') \end{pmatrix}$$

Given a set of samples, $D = \{(s_{d_i}, a_{d_i}, r_{d_i}, s'_{d_i}) \mid i = 1, 2, \dots, L\}$, we can construct approximate versions of $\boldsymbol{\Phi}$, $\mathbf{P}^\pi\boldsymbol{\Phi}$, and $\mathcal{R}$ as follows :

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

These approximations can be thought of as first sampling rows from $\boldsymbol{\Phi}$ and then, conditioned on these samples, as sampling terms from the summations in the corresponding rows of $\mathbf{P}^\pi\boldsymbol{\Phi}$ and $\mathcal{R}$. The sampling distribution from the summations is governed by the underlying dynamics $(P(s, a, s'))$ of the process as the samples in D are taken directly from the MDP. Therefore, $\mathbf{A}$ and b can be approximated as

$$\widehat{\mathbf{A}} = \widehat{\boldsymbol{\Phi}}^\intercal(\widehat{\boldsymbol{\Phi}} - \gamma \widehat{\mathbf{P}^\pi\boldsymbol{\Phi}}) \qquad \text{and} \qquad \widehat{b} = \widehat{\boldsymbol{\Phi}}^\intercal \widehat{\mathcal{R}} \ .$$

These equations lead 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 π, a sample (s, a, r, s') contributes to the approximation according to the following update equation :

$$\widehat{\mathbf{A}} \leftarrow \widehat{\mathbf{A}} + \phi(s, a)\Big(\phi(s, a) - \gamma\phi(s', \pi(s'))\Big)^\intercal \qquad \text{and} \qquad \widehat{b} \leftarrow \widehat{b} + \phi(s, a) r \ .$$

With uniformly distributed samples over pairs of states and actions (s, a), the approximations $\widehat{\mathbf{A}}$ and $\widehat{b}$ are consistent approximations of the true $\mathbf{A}$ and b (scaled by a constant) and the solution $\widehat{w}^\pi$ will converge to the true solution w^π.

We call this algorithm LSQ [7] due to its similarity to LSTD. LSQ learns the state-action value function of a fixed policy. However, unlike LSTD, it computes Q functions and does not expect the data to come from any particular Markov chain. LSQ can use the same set of samples to compute Q values for any policy. The policy merely determines which $\phi(s', \pi(s'))$ is added to $\widehat{\mathbf{A}}$ for each sample.

3.3 LSQL: Least-Squares Q-Learning

In our early work [8,9] we proposed a direct extension of LSQ to control problems in the spirit of Q-learning. Recall that Q-learning uses the current approximation to derive an estimate of the (maximum) value of the resulting state. Applying the same idea to modify LSQ, we arrived at the following update equations for any sample (s, a, r, s'):

$$\widehat{\mathbf{A}}^{(t+1)} \leftarrow \mu \widehat{\mathbf{A}}^{(t)} + \phi(s,a)\phi(s,a)^\intercal \ , \quad \widehat{b}^{(t+1)} \leftarrow \mu \widehat{b}^{(t)} + \phi(s,a)\Big(r + \gamma \max_{a'} \phi(s',a')^\intercal w^{(t)}\Big) \ .$$

The weight vector $w^{(t)}$ at each step t is the solution of the system $\widehat{\mathbf{A}}^{(t)} w^{(t)} = \widehat{b}^{(t)}$. Essentially, the nonlinear term introduced by the maximization operator is explicitly computed using the current estimates and becomes part of the right hand side of the system. Unlike Q-learning, the effect of a sample does not fade out because of the absence of a learning rate. The parameter $\mu \in (0,1]$ is an exponential windowing factor and is used to discount the oldest, and thus most inaccurate, entries in $\widehat{\mathbf{A}}$ and $\widehat{b}$.

Although LSQL is a reasonable and intuitive extension, it has some limitations. The use of the current estimates introduces significant bias in the approximation, especially in the early steps when the estimates are inaccurate. Also, samples are not used so efficiently since they are discarded after one use. Even if they were stored and reused, numerous passes are required before the inaccurate information entered early in the matrices is replaced by more accurate estimates.

3.4 LSPI: Least-Squares Policy Iteration

LSQ does not suffer from the problems of LSQL because its equations are strictly linear, however it can learn value functions for fixed policies only. Thus, LSQ can be integrated into an approximate policy iteration procedure (performing the value determination step) for solving learning control problems . This is the key insight behind the Least-Squares Policy Iteration (LSPI) algorithm [7]. Note that this is not the same as using LSTD in a policy iteration algorithm. LSQ approximations are not biased by the stationary distribution, since samples can be collected arbitrarily and their distribution can be potentially controlled. More importantly, the policy improvement step of the policy iteration can be realized automatically without ever explicitly representing the policy and without any sort of model. Since LSQ computes Q functions, the improved policy $\pi^{(t+1)}$ is simply the greedy policy over the Q function learned in the previous iteration:

$$\pi^{(t+1)}(s) = \arg\max_a \widehat{Q}^{\pi^{(t)}}(s,a) = \arg\max_a \phi(s,a)^\intercal w^{\pi^{(t)}} \ .$$

In this sense the improved (greedy) policy is represented implicitly by a finite set of parameters ($w^{\pi^{(t)}}$) and can be determined on demand for any given state as shown above. To close the loop, we require that LSQ performs this maximization to find $\pi^{(t)}(s')$ for each s' in the data set when constructing the $\widehat{\mathbf{A}}$ matrix for a policy $\pi^{(t)}$. The LSPI algorithm is summarized in Figure 1.

```
**LSPI** ($k$, $\phi$, $\gamma$, $\epsilon$, $\pi_0$, $D_0$)

    // $k$   : Number of basis functions
    // $\phi$   : Basis functions
    // $\gamma$   : Discount factor
    // $\epsilon$   : Stopping criterion
    // $\pi_0$  : Initial policy, given as $w_0$ (default: $w_0 = 0$)
    // $D_0$  : Initial set of training samples, possibly empty

    $D = D_0$
    $\pi' = \pi_0$          // In essence, $w' = w_0$

    repeat
        Update $D$ (optional)              // Add/remove samples, or leave unchanged
        $\pi = \pi'$                          // $w = w'$
        $\pi' = $ LSQ ($D$, $k$, $\phi$, $\gamma$, $\pi$)      // $w' = $ LSQ ($D$, $k$, $\phi$, $\gamma$, $w$)
    until ($\pi \approx \pi'$)                   // that is, ($||w - w'|| < \epsilon$)

    return $\pi$                             // return $w$
```

Fig. 1. The LSPI algorithm.

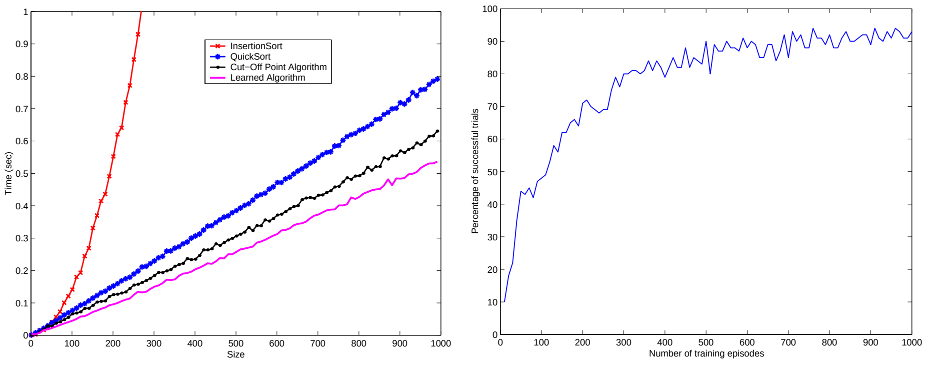
4 Experimental Results

4.1 Algorithm Selection

Algorithm selection [13] is the following decision problem: given a set of algorithms for a problem, dynamically choose the best algorithm for any instance of the problem, i.e. the algorithm that minimizes the expected total execution time on a target machine. The problem becomes more challenging with recursive algorithms in the set. A sub-instance generated during a recursive call gives rise to a new algorithm selection problem; any algorithm in the set can be chosen to solve it. We call this sequential decision problem *recursive algorithm selection* [8], since the entire sequence (or tree) of decisions has to be optimized. Uncertainty in algorithm selection stems from the input distribution, the inner workings of the algorithms (e.g. randomized algorithms), and the hardware characteristics.

We can formulate the problem as a kind of MDP. The state of the process consists of a set of instance features, such as problem size. The actions are the different algorithms we can choose from. Non-recursive algorithms are terminal in that they solve the instance completely (terminal state). Recursive algorithms create subinstances and therefore cause (non-deterministic) transitions to other states. The immediate cost of a decision is the real time taken for executing the selected algorithm on the current instance, excluding time taken in recursive calls. Thus, the total (undiscounted) cost during an episode is the total time taken to solve that particular instance. The goal is to find a policy that minimizes the expected total cost/time. This process differs from a standard MDP in that it allows one-to-many state transitions (multiple recursive calls at one level).

We used LSQL to learn good policies for the following problems: order-statistic selection [8], sorting [8], and branching in satisfiability [9]. For sorting, we combined InsertionSort and QuickSort using the array size n as the only state feature. The linear approximator included a block of three terms (n, $n \log_2 n$, and



(a) (b)

Fig. 2. Results on: (a) algorithm selection for sorting; (b) the inverted pendulum.

n^2) repeated for each action, thus a total of six basis functions. In effect, each action had its own separate set of weights over the same set of basis functions. After training, the learned policy was tested against the individual algorithms and against an empirical cut-off point algorithm. Averaged results are shown in Figure 2 (a). For sorting, in particular, it is easy to derive the transition model and use a model-based approach to obtain even better selection policies [10].

For satisfiability, we considered the problem of selecting among seven heuristic branching rules at each branching point of a DPLL procedure for the SAT or #SAT problem [9]. The state was the number of free variables n at the current node and the immediate cost was the number of nodes expanded between the current and the next branching nodes. With this definition, the total undiscounted cost of a complete episode is the total number of nodes expanded during the DPLL run. Since the Q function was expected to be exponential in n, we used a polynomial in n of degree 7 (with no constant term) to approximate the logarithm of the Q function separately for each action (49 basis functions total). We used LSQL to learn selection policies on different classes of #SAT problems. The learned policies performed as well as the best of the individual heuristics, and in one class of problems significantly better. In all cases, the learned policies were significantly better than the purely randomized policy.

4.2 Inverted Pendulum

The *inverted pendulum* problem is to balance a pendulum of unknown length and mass at the upright position. The state space is continuous and consists of the vertical angle and the angular velocity of the pendulum. The (nonlinear) dynamics of the system are described in [16]. There are three force actions, $\mathcal{A} = \{-50, 0, +50\}$, but the actual input u to the system is noisy; $u = a + 10n$, where $a \in \mathcal{A}$ and n is a Gaussian noise term. The simulation step is 0.1 seconds. The agent receives zero reward as long as the angle of the pendulum does not exceed $\pi/2$ in absolute value. An angle greater than $\pi/2$ signals the end of the episode and a penalty of -1. The discount factor of the process is 0.9.

We used a set of 30 basis functions (10 for each action) to approximate the value function. These 10 basis functions include a constant term and 9 radial basis functions (Gaussians with $\sigma^2 = 1$) arranged in a 3×3 grid ($\{-\pi/4,\ 0,\ +\pi/4\} \times \{-1,\ 0,\ +1\}$) over the 2-dimensional state space. Training samples were collected from "random episodes", i.e., starting in a random state close to the upright position and following a purely random policy. Figure 2 (b) shows the performance of controllers learned by LSPI. Each (successful) episode was allowed to run for a maximum of 3000 steps (5 minutes) of continuous balancing. LSPI returned very good policies given only a few hundred training episodes.

4.3 Bicycle Balancing and Riding

The goal in the bicycle problem [12] is to learn to balance and ride a bicycle to a target position located 1 km away from the starting location. Initially, the bicycle's orientation is at an angle of 90° to the goal. The state description is a six-dimensional real-valued vector $(\theta, \dot{\theta}, \omega, \dot{\omega}, \ddot{\omega}, \psi)$, where θ is the angle of the handlebar, ω is the vertical angle of the bicycle, and ψ is the angle of the bicycle to the goal. The actions are the torque τ applied to the handlebar (discretized to $\{-2, 0, +2\}$) and the displacement of the rider υ (discretized to $\{-0.02, 0, +0.02\}$). In our experiments, actions are restricted to be either τ or υ (or nothing) giving a total of 5 actions. A shaping reward signal was used to learn both tasks at once. The agent receives a reward equal to the net change in the square of the vertical angle and a reward equal to 1% of the net change (in meters) in the distance to the goal. These two rewards are combined additively at each time step. The discount factor is 0.8. The noise in the system is a uniformly distributed term in $[-0.02, +0.02]$ added to the displacement component of the action. The dynamics of the bicycle are based on the model described in [12] and the time step of the simulation is set to 0.01 seconds.

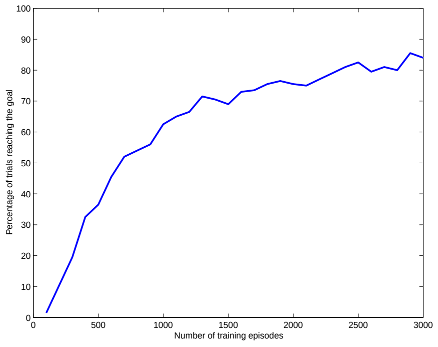
The state-action value function $Q(s, a)$ for a fixed action a is approximated by a linear combination of 20 basis functions:

$$(\ 1,\ \omega,\ \dot{\omega},\ \omega^2,\ \dot{\omega}^2, \omega\dot{\omega},\ \theta,\ \dot{\theta},\ \theta^2,\ \dot{\theta}^2,\ \theta\dot{\theta},\ \omega\theta,\ \omega\theta^2,\ \omega^2\theta,\ \psi,\ \psi^2,\ \psi\theta,\ \bar{\psi},\ \bar{\psi}^2,\ \bar{\psi}\theta\ )\ ,$$

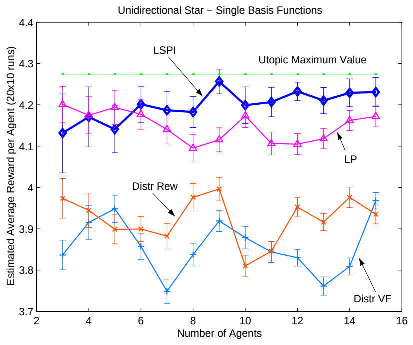
where $\bar{\psi} = \pi - \psi$ for $\psi > 0$ and $\bar{\psi} = -\pi - \psi$ for $\psi < 0$. Note that the state variable $\ddot{\omega}$ is completely ignored. This block of basis functions is repeated for each of the 5 actions, giving a total of 100 basis functions and weights. Training data were collected by initializing the bicycle to a random state around the equilibrium position and running small episodes of 20 steps each using a purely random policy. LSPI was applied on training sets of different sizes and the average performance is shown in Figure 3 (a). Successful policies usually reached the goal in approximately 1 km total, near optimal performance.

4.4 Multiagent Learning: The SysAdmin Problem

In multiagent domains, multiple agents must coordinate their actions so as to maximize their joint utility. Such systems can be viewed as MDPs where the



(a)



(b)

Fig. 3. Results on: (a) bicycle balancing and riding; (b) the SysAdmin problem.

“action” is the joint action and the reward is the total reward for all of the agents. Although, the action space can be quite large, *Collaborative action selection* [3] is a method that allows multiple agents to efficiently determine the jointly optimal action with respect to an (approximate) factored value function using a simple message passing scheme. This joint value function is a linear combination of local value functions, each of which relates only to some parts of the system controlled by a small number of agents. Extending LSPI to multiagent learning in such domains is straightforward. LSPI can learn the coefficients for the factored value function and the improved policy will be defined implicitly by the learned Q-function. However, instead of enumerating the exponentially many actions to find the maximizing action, the collaborative action selection mechanism is used to determine efficiently the policy at any given state.

The *SysAdmin* problem [3] consists of a network of n machines connected in a chain, ring, star, ring-of-rings, or star-and-ring topology. The state of each machine is described by its status (good, faulty, dead) and its load (idle, loaded, process successful). Jobs can be executed on good or faulty machines (job arrivals and terminations are stochastic), but a faulty machine will take longer to terminate. 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. Machines fail stochastically and they are also influenced by their neighbors. Each machine is also associated with a rebooting agent. 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 to maximize the total reward for the system. The discount factor is 0.95. The SysAdmin problem has been studied in [3], where the model of the process is assumed to be available as a factored MDP. 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 close to the theoretical optimal and significantly better compared to policies learned by the Distributed Reward (DR) and Distributed Value Function (DVF) algorithms [14].

In our work, we assume that no model is available and we applied LSPI to learn rebooting policies [4]. To make a fair comparison, we used comparable sets of basis functions. For n machines in the network, we experimentally found that about $600n$ samples are sufficient for LSPI to learn a good policy. The samples were collected by a purely random policy. Figure 3 (b) shows the results obtained by LSPI on the star topology compared to the results of LP, DR, and DVF as reported in [3]. In both cases, LSPI learns very good policies comparable to the LP approach, 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.

4.5 Two-Player Zero-Sum Markov Games

A two-player zero-sum Markov game is defined by a set of states $\mathcal{S}$ and two sets of actions, $\mathcal{A}$ and $\mathcal{O}$, one for each player. In each state, the two players take actions simultaneously, they receive a reward that depends on the current state and their actions, and the game makes a stochastic transition to a new state. The two players have diametrically opposed goals; one is trying to maximize the total cumulative reward, whereas the other is trying to minimize it. Optimality can be defined independently of the opponent in the minimax sense: maximize your total reward in the worst case. Unlike MDPs, the minimax-optimal policy for a Markov game need not be deterministic. Littman [11] has studied Markov games as a framework for multiagent RL by extending tabular Q-learning to a variant called minimax-Q.

We tried to apply LSPI to the same kind of problems. Given an approximate value function $\widehat{Q}(s,a,o)$, the implied policy at any given state s is a probability distribution π_s over actions defined as

$$\pi_s = \arg \max_{\pi_s \in \mathrm{PD}(\mathcal{A})} \min_{o \in \mathcal{O}} \sum_{a \in \mathcal{A}} \pi_s(a) \widehat{Q}(s,a,o) \ ,$$

where a is the action of our agent and o is the opponent's action. π_s can be found by solving a linear program [11]. Given that the policy is stochastic, the update equations of LSQ within LSPI have to be modified so that the distribution over possible next actions is taken into account:

$$\widehat{\mathbf{A}} \leftarrow \widehat{\mathbf{A}} + \phi(s,a,o)\Big(\phi(s,a,o) - \gamma \sum_{a' \in \mathcal{A}} \pi_{s'}(a')\phi(s',a',o')\Big)^{\intercal} \ , \ \widehat{b} \leftarrow \widehat{b} + \phi(s,a,o)r \ ,$$

for any sample (s,a,o,r,s'). The action o' is the minimizing opponent's action in computing $\pi_{s'}$. In our preliminary experiments on the simplified one-on-one soccer game [11], LSPI was able to learn very good policies using only about $10,000$ samples. This is a fraction of the $1,000,000$ samples required by tabular minimax-Q. Further, with the use of basis functions that capture important features of the game (e.g., scaled distances to the goals and the opponent) for approximating the value function, we have been able to scale to grid sizes much bigger than the original (5×4) grid. We are currently investigating team-based Markov games and the use of coordinated action selection in conjunction with LSPI for efficient multiagent learning in team-based competitive domains.

4.6 Tetris

Tetris is a popular tiling video game. Although the model of the game is rather simplistic and known in advance, the state-action space is so big ($\approx 10^{61}$ states and ≈ 40 actions) that one has to rely on approximation and learning techniques to find good policies. We used 10 basis functions over (s, a) pairs to capture features of state s and the one-step effects of playing action a in s: the maximum height in the current board, the total number of "holes", the sum of absolute height differences between adjacent columns, the mean height, and the change of these quantities in the next step, plus the change in score and a constant term. That results in a single set of 10 weights for all actions.

In our preliminary results, policies learned by LSPI using about $10,000$ samples achieve average score between $1,000$ and $3,000$ points per game. The training samples were collected using a hand-crafted policy that scores about 600 points per game (the random policy rarely scores any point). Knowledge about the model was incorporated in LSPI to improve the approximation: for each sample, instead of considering just the sampled next state in the update equation, we considered a sum over all possible next states appropriatelly weighted according to the transition model.

Our results compare favorably with the results of $\lambda-$policy iteration on Tetris [1], but there are significant differences in the two approaches. $\lambda-$policy iteration collects new samples in each iteration and learns the state value function; it uses the model for greedy action selection over the learved function, and the iteration does not finally converge. On the contrary, LSPI collects samples only once at the very beginning and learns the state-action value function; it uses the model only to improve the approximation, and converges in about 10 iterations. In both cases, the learned players exhibit big variance in performance.

5 Discussion and Conclusion

We presented an overview of our research efforts towards using least-squares methods in reinforcement learning control problems. The key advantages of least-squares methods is the efficient use of samples and the simplicity of the implementation. In all the domains we tested, our algorithms were able to learn very good policies using only a small number of samples compared to conventional learning approaches, such as Q-learning. Moreover, the algorithms required little or no modification in each case. There are also many exciting avenues to explore further: How are the basis functions chosen? What is the effect of the distribution of the training samples? Can we use projection reweighting methods to make LSPI amenable to even "bad" data sets? These are some of the many open questions on our research agenda. In any case, we believe that algorithms like LSPI can easily be good first-choice candidates for many reinforcement learning control problems.

We would like to thank C. Guestrin, D. Koller, and U. Lerner for helpful discussions. The first author would also like to thank the Lilian-Boudouri Foundation in Greece for partial financial support.

References

1. D. Bertsekas and J. Tsitsiklis. *Neuro-Dynamic Programming.* Athena Scientific, Belmont, Massachusetts, 1996.
2. Steven J. Bradtke and Andrew G. Barto. Linear least-squares algorithms for temporal difference learning. *Machine Learning*, 22(1/2/3):33–57, 1996.
3. Carlos Guestrin, Daphne Koller, and Ronald Parr. Multiagent planning with factored MDPs. In *Proceeding of the 14th Neural Information Processing Systems (NIPS-14)*, Vancouver, Canada, December 2001.
4. Carlos Guestrin, Michail G. Lagoudakis, and Ronald Parr. Coordinated reinforcement learning. In *Proceedings of the 2002 AAAI Spring Symposium Series: Collaborative Learning Agents*, Stanford, CA, March 2002.
5. Leslie P. Kaelbling, Michael L. Littman, and Andrew W. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.
6. Daphne Koller and Ronald Parr. Policy iteration for factored MDPs. In Craig Boutilier and Moisés Goldszmidt, editors, *Proceedings of the 16th Conference on Uncertainty in Artificial Intelligence (UAI-00)*, pages 326–334, San Francisco, CA, 2000. Morgan Kaufmann Publishers.
7. Michail Lagoudakis and Ronald Parr. Model free least squares policy iteration. In *Proceedings of the 14th Neural Information Processing Systems (NIPS-14)*, Vancouver, Canada, December 2001.
8. Michail G. Lagoudakis and Michael L. Littman. Algorithm selection using reinforcement learning. In Pat Langley, editor, *Proceedings of the Seventeenth International Conference on Machine Learning*, pages 511–518. Morgan Kaufmann, San Francisco, CA, 2000.
9. Michail G. Lagoudakis and Michael L. Littman. Learning to select branching rules in the dpll procedure for satisfiability. In Henry Kautz and Bart Selman, editors, *Electronic Notes in Discrete Mathematics (ENDM), Vol. 9, LICS 2001 Workshop on Theory and Applications of Satisfiability Testing.* Elsevier Science, 2001.
10. Michail G. Lagoudakis, Michael L. Littman, and Ronald Parr. Selecting the right algorithm. In Carla Gomes and Toby Walsh, editors, *Proceedings of the 2001 AAAI Fall Symposium Series: Using Uncertainty within Computation*, Cape Cod, MA, November 2001.
11. Michael L. Littman. Markov games as a framework for multi-agent reinforcement learning. In *Proceedings of the Eleventh International Conference on Machine Learning*, pages 157–163, San Francisco, CA, 1994. Morgan Kaufmann.
12. J. Randløv and P. Alstrøm. Learning to drive a bicycle using reinforcement learning and shaping. In *Proceedings of The Fifteenth International Conference on Machine Learning*, Madison, Wisconsin, July 1998. Morgan Kaufmann.
13. John R. Rice. The algorithm selection problem. *Advances in Computers*, 15:65–118, 1976.
14. J. Schneider, W. Wong, A. Moore, and M. Riedmiller. Distributed value functions. In *Proceedings of The Sixteenth International Conference on Machine Learning*, Bled, Slovenia, July 1999. Morgan Kaufmann.
15. R. Sutton and A. Barto. *Reinforcement Learning: An Introduction.* MIT Press, Cambridge, MA, 1998.
16. K. Wang, H. Tanaka and M. Griffin. An approach to fuzzy control of nonlinear systems: Stability and design issues. *IEEE Transactions on Fuzzy Systems*, 4(1):14–23, 1996.
17. Christopher J. C. H. Watkins. *Learning from Delayed Rewards.* PhD thesis, King's College, Cambridge, UK, 1989.