Approximate Policy Iteration with Linear Action Models

Hengshuai Yao, Csaba Szepesvári

Reinforcement Learning and Articial Intelligence Laboratory
Department of Computing Science
University of Alberta, Edmonton, AB Canada T6G 2E8

Abstract

In this paper we consider the problem of finding a good policy given some batch data. We propose a new approach, LAM-API, that first builds a so-called linear action model (LAM) from the data and then uses the learned model and the collected data in approximate policy iteration (API) to find a good policy. A natural choice for the policy evaluation step in this algorithm is to use least-squares temporal difference (LSTD) learning algorithm. Empirical results on three benchmark problems show that this particular instance of LAM-API performs competitively as compared with LSPI, both from the point of view of data and computational efficiency.

Introduction

Reinforcement learning is concerned with finding a good controller, or policy given some data to maximize a long-term performance measure [Sutton and Barto, 1998; Bertsekas and Tsitsiklis, 1996]. Reinforcement learning problems can be interactive or batch, depending on whether the learning system can interact with the system that is to be controlled [Szepesvári, 2010]. In this paper we consider batch reinforcement learning problems. We assume that a previous controller generates data in the form of a series of trajectories that are composed of a sequence of observations, actions and immediate rewards triples. We also assume that a method that extracts real-valued features based on the observations is given. The problem is then to learn a policy that maps observable quantities to actions in such a way that when it is used to interact with the system it would achieve a good performance as measured in terms of the total discounted sum of rewards received.

Algorithms developed for this problem should be data efficient (learn from "little data") while being computationally efficient at the same time. Needless to say, they should also be reliable: at minimum their performance should reliably improve as the samples become infinitely many and/or as the set of features becomes "better". The problem has been tackled previously by a number of methods including policy search methods and value function-based methods (for recent overviews, see Szepesvári 2010; Wiering and Otterlo 2012). One of the prominent methods is *least-squares policy iteration* (LSPI) [Lagoudakis and Parr, 2003],

which belongs to the family of approximate policy iteration (API) algorithms. LSPI generates a sequence of action-value functions using the so-called *least-squares temporal difference learning* (LSTD) algorithm [Bradtke and Barto, 1996; Boyan, 2002] in each iteration to approximately evaluate the policy that is greedy with respect to the previous action-value function. While Lagoudakis and Parr [2003] has shown that LSPI is a competitive algorithm, more recently a number of papers studied the theoretical performance of LSPI (see [Buşoniu, Lazaric, Ghavamzadeh, Munos, Babuška, and Schutter, 2012] and the references therein). These results essentially show that under some technical (but stringent) conditions on the Markovian Decision Process (MDP), provided that the features are "strong enough", LSPI is guaranteed to return a near-optimal policy. In particular, the gap between the performance of the returned policy and the optimal policy can be bounded in terms of the sum of an "approximation error term" (characterizing how well the features "resolve" the value functions of the relevant policies), a "variance term" (that decreases as the number of samples increases), and a term that converges to zero exponentially fast with an increasing number of iterations.

The so-called Bellman error of the approximate value functions returned by LSTD decomposes into the error with which the reward function and the "next state-features" are predicted using least-mean squares predictions built on the linear combination of features [Parr, Li, Taylor, Painter-Wakefield, and Littman, 2008; Sutton, Szepesvári, Geramifard, and Bowling, 2008] (Section 2.2.1 of [Szepesvári, 2010] extends this to LTSD(λ)). This shows that the smaller the prediction error of the rewards and the next state-features is, the better the value function returned by LSTD will approximate the true value function of the policy evaluated. This suggests an alternative to LSPI. We can build a linear model for the rewards and next states for each action based on the available features and then find a good policy based on this model. This is the approach that we study in this paper. Since the model tries to capture the effects of the actions with a linear combination of a set of features, we call it a *linear action model*, or "LAM". Given the model, the second phase can use any number of methods to find a policy whose performance would be near-optimal *in the model*. In this paper, we study using approximate policy iteration where the

policies encountered are evaluated using LSTD. The difference to LSPI of this second phase is twofold. First, the data fed to approximate policy iteration comes partly from the model. Second, since we have access to a model, it suffices to build state-value functions.

Background

The purpose of this section is to introduce the necessary concepts and notation. We start by reviewing concepts related to MDPs which define the framework we use to define our learning problem. First a few notes on the conventions used in this paper: We use $\{\cdot\}^\top$ to denote the transpose and all (non-transposed) vectors will denote column vectors.

We define a finite *MDP* as a 5-tuple, $(\mathbb{S}, \mathbb{A}, \mathcal{P}^\mathbb{A}, \mathcal{R}^\mathbb{A}, \gamma)$, where $\mathbb{S}$ is the finite state space, $\mathbb{A}$ is the finite action space, $\mathcal{P}^\mathbb{A}$ is a transition model with $\mathcal{P}^a(s, s')$ being the probability of transitioning to state s' after taking action a at state s, $\mathcal{R}^\mathbb{A}$ is a reward model with $\mathcal{R}^a(s, s')$ being the reward of the state transitioning, and $\gamma \in (0, 1)$ is a discount factor. That the state space is finite is assumed for simplicity, while the assumption that the action space is finite (and "small") will be exploited by the algorithms which need to solve some maximization problems over the action space. An MDP defines a controllable system: The controller needs to choose the actions based on the information available to it. The purpose of the rewards is to define a (partial) ordering over the space of possible controllers.

A *policy* π assigns probabilities to each action for each state. We will denote by $\pi(s, a)$ the probability assigned to action a in state s by policy π. A policy can be used as a closed-loop controller: When visiting state s, the action to be taken is sampled from $\pi(s, \cdot)$. The value of a state s under a policy π is the expected total discounted reward received if the policy is followed when starting in state s:

$$V^\pi(s) = E_\pi \left\{ \sum_{t=0}^\infty \gamma^t r_t | s_0 = s \right\}.$$

Here r_t is the reward received by the agent at time t, and E_π is the expectation taken with respect to the distribution generated by policy π. The function V^π is called the *state value function*.

A policy that achieves the best possible value in each state is called an *optimal policy*. Thus,

$$V^{\pi^*}(s) = \max_\pi V^\pi(s)$$

holds for all $s \in \mathbb{S}$ if π^* is an optimal policy.

The goal of reinforcement learning is to find a (near) optimal policy either by interacting with an MDP, or based on some interaction traces. In this paper we consider this second case, i.e., the case of batch reinforcement learning. In particular, we assume that data is available in the form of a list of 4-tuples of elementary transition snippets,

$$(\langle \phi_i, a_i, \phi'_{i+1}, r_i \rangle; \quad i = 1, \dots, n), \tag{1}$$

where $\phi_i = \phi(s_i)$, $\phi'_{i+1} = \phi(s'_{i+1})$, $s_i, s'_{i+1} \in \mathbb{S}$, $a_i \in \mathbb{A}$, $s'_{i+1} \sim \mathcal{P}^{a_i}(s_i, \cdot)$ and $r_i = \mathcal{R}^{a_i}(s_i, s'_{i+1})$. Here, $\phi : \mathbb{S} \to \mathbb{R}^d$ is a so-called "feature extraction function" (d is the number of features). The problem is to learn a policy that depends on the states only through their features.

Given the transition snippets of a fixed policy π, LSTD aims at finding a linear-in-the-features approximation $V_\theta(s) = \theta^\top \phi(s)$ (for $\forall s \in \mathbb{S}$) to the value function V^π. For this, it builds a matrix $d \times d$ matrix A and d-dimensional vector b

$$A = \sum_{i=1}^n \phi_i(\gamma \phi'_{i+1} - \phi_i)^\top, \quad b = \sum_{i=1}^n \phi_i r_i$$

and then finds $\theta = -A^{-1}b$. The matrix A and vector b can be built incrementally.

The LSTD algorithm naturally leads to the idea of LSPI due to Lagoudakis and Parr [2003]. LSPI is an instance of approximate policy iteration and thus let us start by introducing policy iteration. To do this, first, let us define the action-value functions of policies. An *action-value function* Q^π of a policy π maps state-action pairs to reals ($Q^\pi : \mathbb{S} \times \mathbb{A} \to \mathbb{R}$) and for a given state-action pair (s, a), $Q^\pi(s, a)$ equals to the total expected discounted reward given that the decision process starts in the state s, the first action taken is a from which point on all the actions are taken according to the policy π.

Policy iteration is an iterative method that produces a sequence of policies [Howard, 1960]. In a finite MDP after a (small) finite number of steps an optimal policy is obtained [Ye, 2010], while in infinite MDPs the value functions of the policies produced converges at a geometric rate to the optimal value function, i.e., the performance of policies improves rapidly. In iteration k, given a policy π_k, policy iteration first computes the action-value function of π_k, $Q_k := Q^{\pi_k}$. The next policy π_{k+1} is then selected to be the greedy policy w.r.t. Q_k. This means, that in every state s, π_{k+1} selects an action amongst the ones that maximize $Q_k(s, \cdot)$ with probability one. In policy iteration often one selects deterministic policies, i.e., policies where $\pi(s, \cdot)$ concentrates on a single state for every state $s \in \mathbb{S}$. For simplicity, let us also consider such policies. By abusing notation, such deterministic policies will be identified with functions that map states to actions.

In LSPI the exact policy evaluation step is replaced by an approximate one. For this, LSTD is extended to the LSTDQ algorithm. This algorithm approximates Q^{π_k} by $Q_\vartheta(s, a) = \vartheta_k^\top \psi(s, a)$, where $\psi : \mathbb{S} \times \mathbb{A} \to \mathbb{R}^{d'}$ is now a feature extraction method that assigns features to state-action pairs, and ϑ_k is the value of the policy parameter vector $\vartheta \in \mathbb{R}^{d'}$ at iteration k. In the case of finite actions, assuming without loss of generality that $\mathbb{A} = \{1, \dots, |\mathbb{A}|\}$, a natural choice is to use $\psi(s, a) = [0, \dots, 0, \phi(s)^\top, 0, \dots, 0]^\top \in \mathbb{R}^{d|\mathbb{A}|}$ where $\phi(s)^\top$ appears in the ath block of length d of the $d|\mathbb{A}|$-dimensional vector. Defining $\psi_i = \psi(s_i, a_i)$, $\psi'_{i+1} = \psi(s'_{i+1}, a'_{i+1})$ for some action a'_{i+1}, $i = 1, 2, \dots, n$, given a policy π_k, LSTDQ computes $A = \sum_{i:a'_{i+1} = \pi_k(s'_{i+1})} \psi_i(\gamma \psi'_{i+1} - \psi_i)^\top$, $b = \sum_i \psi_i r_i$ and solves for $\vartheta = -A^{-1}b$. [1] At iteration k, for any state s_i, LSPI sets $a'_{i+1} = \arg\max_{a \in \mathbb{A}} Q_{k-1}(s'_{i+1}, a)$.

---

[1] Here we abused the notion for A and b. Caution that they are not the same as LSTD's.

Algorithm 1 The algorithm of learning LAM.

Input: a data set, $\mathcal{S} = (\langle \phi_i, a_i, \phi'_{i+1}, r_i \rangle; i = 1, \ldots, n)$
Output: a LAM, $(\langle F^a, f^a \rangle)_{a \in \mathbb{A}}$.
Initialize H^a, E^a and e^a for all a
for $i = 1, 2, \ldots, d$ **do**
    $a = a_i$
    Update LAM structures of a:
        $H^a = H^a + \phi_i \phi_i^\top$
        $E^a = E^a + \phi'_{i+1} \phi_i^\top$
        $e^a = e^a + \phi_i r_i$
end
for all a, set
    $F^a = E^a (H^a)^{-1}$
    $f^a = (H^a)^{-1} e^a$

Linear Action Models

In this section, we define linear action models (LAMs). Next, we give the form of a least-squares method to learn LAMs, which is followed by an API algorithm that we use in the experimental section to find a good policy given a LAM. In the rest of the paper we fix the feature extraction function ϕ.

Definition 1 *A* linear model *of an action* $a \in \mathbb{A}$ *is a pair* $\langle F^a, f^a \rangle$, *where* $F^a \in \mathbb{R}^{d \times d}$ *is a* $d \times d$ *matrix and* $f^a \in \mathbb{R}^d$ *is a* d-*dimensional vector. A* linear action model $(\langle F^a, f^a \rangle)_{a \in \mathbb{A}}$ *is a list of linear models, one for each action* $a \in \mathbb{A}$.

The idea is that for a given action a and any given state $s \in \mathbb{S}$, $F^a \phi(s)$ estimates the expected value of the feature vector of the state $s' \sim \mathcal{P}^a(s, \cdot)$, while $(f^a)^\top \phi(s)$ estimates the reward associated with the transition (s, a, s'). For a good linear model we expect that with $s' \sim \mathcal{P}^a(s, \cdot)$,

$$F^a \phi(s) \approx E[\phi(s')] \quad \text{and} \quad (f^a)^\top \phi(s) \approx E[\mathcal{R}^a(s, s')].$$

The ideas in the paper extend easily to when one models policies or even options. In fact, the idea of using linear models originates from the paper [Sutton et al., 2008] where linear models of both policies and actions have been built and used in a Dyna-like algorithm that aimed at learning a good policy in an interactive learning scenario. Linear models of options (temporally extended actions) have been considered in a more recent paper of Sorg and Singh [2010] that we have learned about after the submission of the paper. The relationship of the present paper to these prior works will be discussed later.

Learning a LAM

Assume that we are given a dataset in the form of transition snippets as in (1). An obvious way of learning a model of some action $a \in \mathbb{A}$ is to filter the data for the snippets when $a = a_i$ and estimate F^a, f^a using (regularized) least-squares:

$$F^a = \arg \min_{F \in \mathbb{R}^{d \times d}} \sum_{i : a_i = a} \| \phi'_{i+1} - F \phi_i \|_2^2 + \lambda_1 \| F \|_{\mathbb{F}}^2 \quad (2)$$

$$f^a = \arg \min_{f \in \mathbb{R}^d} \sum_{i : a_i = a} \| r_i - f^\top \phi_i \|_2^2 + \lambda_2 \| f \|_2^2. \quad (3)$$

Here $\lambda_1, \lambda_2 > 0$ are regularization coefficients, $\| v \|_2$ denotes the Euclidean norm of vector v, while $\| A \|_{\mathbb{F}} =$ $(\text{trace}(A^\top A))^{1/2}$ denotes the Frobenius norm of matrix A. The purpose of regularization is to avoid degeneracy and improve generalization performance in the case when for some actions the number of transition snippets is small. Note that for simplicity (and computational efficiency) we have chosen 2-norm regularization, however, other forms of regularization are also possible, such as penalizing the ℓ^1-norm of vectors and the nuclear-norm of matrices to promote the sparsity of the coefficients. It is left for future research to study such (and other) alternatives of the least-squares method. In practice, the regularization coefficients for a given action a should and can be set either heuristically based on the number of samples available for the given action, or found using simple trials.

Algorithm 1 shows one way of solving the above least-squares problem. In this algorithm, for an action a, matrix H^a accumulates the Grammian of the features, matrix E^a accumulates the correlation of the current-step and next-step features, and vector e^a accumulates the correlation of the features and the rewards. The method does not require tuning a step-size or repeated training. The computational complexity is (roughly) cubic in the number of features and is linear in the size of the dataset. When the number of features is large, a better alternative might be to use a gradient-like algorithm. This tradeoff is extensively discussed in Section 2.2.3 of Szepesvári [2010].

The Abstracted MDP and Approximate Policy Iteration

Notice that a LAM defines an MDP: the state space is $\mathbb{R}^d$ (the space where the features are embedded into), the action space is $\mathbb{A}$. If action $a \in \mathbb{A}$ is used when the "state" is $\phi \in \mathbb{R}^d$, then the next "state" is $\phi' = F^a \phi$, the reward for the transition is $r = (f^a)^\top \phi$, and the discount factor is γ. We see that in fact the MDP is deterministic. One approach then is to view this MDP as an "abstracted" version of the original MDP and find a near-optimal policy in this MDP. Note that a policy π in this MDP assigns probabilities to the actions for any given vector of $\mathbb{R}^d$. Assuming that π is a policy of the abstracted MDP, at deployment time when the feature $\phi := \phi(s)$ is observed the next action should be sampled from $\pi(\phi, \cdot)$. Thus, π *induces* a policy in the original MDP. A "faithful" abstraction (a good set of features) is expected to have the property that if π is a good policy in the abstracted MDP then the policy induced in the original is guaranteed to enjoy a good performance too. This property has been explored in the context of state-aggregation (a special case of extracting features) by Isaza, Szepesvári, Bulitko, and Greiner [2008]. We leave it for future work to extend this analysis to the present context. When an abstraction is faithful then it makes sense to find a near optimal policy in the abstracted MDP. This could be done with any planning method, though we note in passing that that the abstracted MDP is deterministic presents a special opportunity to design faster methods, such as Hren and Munos [2008].

In this paper, we take a somehow conservative approach. The reason is that we cannot expect the model to be meaningful at points of the space $\mathbb{R}^d$ which are "far away" from the features that are in the dataset (unless one is extremely

---

Algorithm 2 LAM-API with LSTD (LAM-LSTD for short).

Inputs: a list of features $\mathcal{D}$, a LAM $(\langle F^a, f^a \rangle)_{a \in \mathbb{A}}$.
Output: a weight vector θ.
Initialize θ
repeat until θ no longer changes
    for ϕ_i in $\mathcal{D}$ **do**
        Select greedy action:
            $a^* = \arg \max_a \{ (f^a)^\top \phi_i + \gamma \theta^\top F^a \phi_i \}$
        Select model:
            $F^* = F^{a^*}$, $f^* = f^{a^*}$
        Produce prediction for features and rewards:
            $\tilde{\phi}_{i+1} = F^* \phi_i$
            $\tilde{r}_i = (f^*)^\top \phi_i$
        Accumulate LSTD structures:
            $A = A + \phi_i (\gamma \tilde{\phi}_{i+1} - \phi_i)^\top$
            $b = b + \phi_i \tilde{r}_i$
    end
    $\theta = -A^{-1} b$
end

---

lucky). Although nothing prevents the models to generate trajectories that visit such parts of the space $\mathbb{R}^d$ for some policies, the value predictions built for such policies using the model might be very far from their true values.

Algorithm 2 shows the algorithm we propose. The algorithm is an instance of approximate policy iteration that uses value functions that are linear over the features ϕ. Its inputs are the list of features $(\phi_i)_{1 \le i \le n}$ and a LAM $(\langle F^a, f^a \rangle)_{a \in \mathbb{A}}$. Note that although this algorithm can take input from any list of features, in the experiment we used the features from the data set where LAM was learned. By doing this, the method is *grounded in reality*. It is expected that the predictions of the model will be better when the model is applied to the features underlying the actual observations that the model was trained on. Similarly to LSPI, the algorithm uses an implicit representation of policies. However, unlike LSPI it uses state value functions, exploiting that the LAM makes it possible to predict the action-value underlying an action at any point in $\mathbb{R}^d$. In particular, given a parameter vector $\theta \in \mathbb{R}^d$ and any vector $v \in \mathbb{R}^d$, the greedy policy will choose the action that maximizes

$$Q_\theta(v, a) := (f^a)^\top v + \gamma (F^a v)^\top \theta.$$

The inner loop of the algorithm evaluates the resulting policy π_θ by using LSTD on the data that is obtained from the features in the input. In particular, the data is composed of the 3-tuples

$$\langle \phi_i, F^{a_i(\theta)} \phi_i, (f^{a_i(\theta)})^\top \phi_i \rangle, \quad i = 1, \dots, n,$$

where $a_i(\theta) = \pi_\theta(\phi_i)$ is the action chosen by the policy π_θ. Given this data, LSTD gives a new parameter vector θ' such that $V^{\pi_\theta}(s) \approx (\theta')^\top \phi(s)$, $s \in \mathbb{S}$. This process is repeated until the parameter vector does not change or a certain accuracy in the successive change of the parameter is reached. In all the reported experiments of this paper, the parameter vector of our algorithm converges after only a few iterations.

Relation to Previous Work

As mentioned previously, linear action models have been introduced in the context of learning to control when the

learning system interacts with the controlled environment [Sutton et al., 2008]. Learning in an interactive scenario is both easier and harder than learning given a batch of data. The problem is easier because the learning system is given the opportunity to collect new data as needed. On the other hand, if the performance of the learning system is evaluated *while it is learning* then the learning system must avoid poor actions while ensuring that it explores the state space thoroughly, i.e., the system has to balance exploitation and exploration. This is a difficult problem and in its full generality principled solutions are out of the reach of our techniques (see Section 3.2.4 in Szepesvári 2010 for the discussion of existing results). Sutton et al. [2008] took a pragmatic approach to attack this difficult problem. Their algorithm combines a number of ideas: It learns the models simultaneously with using it in a "planning" method that uses TD updates in an optimistic policy iteration algorithm with the learned models. The policy used is the so-called ϵ-greedy policy and the algorithm employs a form of prioritized sweeping in the planning phase.

In comparison, our method is fairly simple, which we view as an advantage. Further, just like any batch method, our method could be adapted to interactive learning (when the batch of data is replaced with the stream of data resulting from the continuous interaction with the controlled system). However, with this we would also need to rely on heuristic techniques at present. In this paper we explored "least-squares" approaches. However, this was possible only because our feature spaces are small. For larger feature spaces one might be forced to replace these methods with incremental, gradient-like methods that use updates whose cost is linear (or less) in the number of features. Luckily, for the purpose of policy evaluation, such algorithms are now available [Sutton et al., 2009b,a].

Sorg and Singh [2010] extended the work of Sutton et al. [2008] to consider options, i.e., temporally extended actions. They also consider the issue of whether the learned models can be used in planning (the "compositionality" issue). In connection to this they make the observation that if both the expected next-state features and the rewards can be predicted with no error from the features of the start-state for any start-state then the models can be arbitrarily composed without any loss of information. An extension of this to the "lossy" case would be of considerable interest. The control learning method of Sorg and Singh [2010] uses Q-learning-like updates that use the option models in place of the "next-state features", which is closer in spirit to Sutton et al. [2008] than to the present paper. Sorg and Singh [2010] demonstrated that their method, as expected, achieves more reward faster than alternatives that do not use options and/or a linear model. As noted earlier, the work presented here could be easily extended to handle options. However, we leave this for future work.

Empirical Results

In this section we present experimental results for the algorithm proposed for three problems of increasing difficulty. The first problem is a 4-state "chain": This small problems allows extensive evaluation and easy to interpret results. The

second problem is the 50-state "chain" problem, which is still rather small (and thus still allows an easy visualization of the results). Besides easy visualization, we include these problems as they were used in a number of previous studies. The last problem studied is the inverted pendulum. We have made the source code of our algorithm and domains available at the authors' webpages.

The Chain Problems

The 4-state Chain. The problem is the chain-walk example used by [Lagoudakis and Parr, 2003]. There are two actions, "Left" (L) and "Right" (R). With probability 0.9, an action leads to a state in the intended direction; with probability 0.1, it leads to a state in the opposite direction. There are 4 states. The discount factor is 0.9. The nonzero reward (one) is given exclusively at the two middle states. The features are, $\phi(s) = [1, s, s^2]^\top$ where $s = 1, 2, 3, 4$. A data set of ten samples of each state-action pair was used (notice this training data has no randomness involved, so the following experiments can be reproduced almost exactly). θ was initialized to 0 for all algorithms.

Figure 1 shows the performance of LAM-LSTD. In the figure, the x-axis shows the states, and the y-axis shows the value of the value function estimates corresponding to the left and right actions. In particular, the function

$$\hat{Q}_L^*(s) = (f^L)^\top \phi(s) + \gamma (F^L \phi(s))^\top \theta,$$

is represented by *blue dash-dot line, marker '•'*; while the function

$$\hat{Q}_R^*(s) = (f^R)^\top \phi(s) + \gamma (F^R \phi(s))^\top \theta,$$

is represented by *red dashed line, marker '+'*. LAM-LSTD found the optimal policy in only 2 iterations. Notice that the success of LAM-API can be somehow predicted by checking the quality of LAM in advance. In fact, because the weight vector θ was initialized to 0, the plot at iteration 0 actually shows the approximated rewards by the features. In Figure 1, the first plot not only shows that the approximated rewards of states 2 and 3 are bigger, but also shows that they are close to each other. For this experiment, the learned linear reward model was $f^L = f^R = [-1.9675, 2.4702, -0.4943]^\top$, and the predicted rewards are $\phi(1)^\top f^L = 0.0084 \approx 0, \phi(2)^\top f^L = 0.9956 \approx 1, \phi(3)^\top f^L = 0.9941 \approx 1, \phi(4)^\top f^L = 0.0039 \approx 0$. The quality of F can be checked before iteration as well, which is omitted here because of limited space. In this experiment, no regularization was used.

We also compared LSPI in the figure. Both algorithms used exactly the same data set. Though the found actions were exactly the same for the two algorithms, the learned (state-value) value functions were different. In particular, LAM-LSTD converged faster than LSPI in the first two iterations. Except that LAM-LSTD uses projected samples, one reason might be, though both algorithms initialized their weight vector to 0, the initial policies were actually different. In particular, LAM-LSTD took advantage of the linear reward model, and the initial state-action value function was
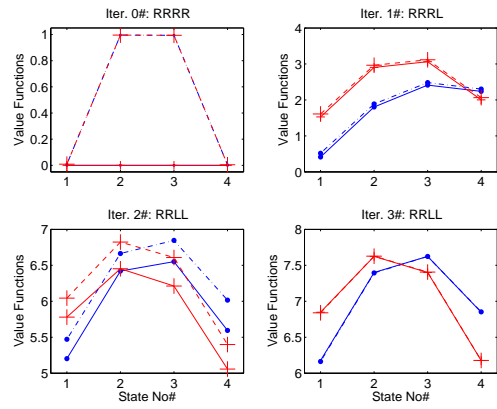

Figure 1: LAM-LSTD (*dashed*) and LSPI (*solid*) for the 4-state chain. The reward is 1 at the two middle states. The value function of action "left" uses the marker '•', while the value function of action "right" uses '+'.
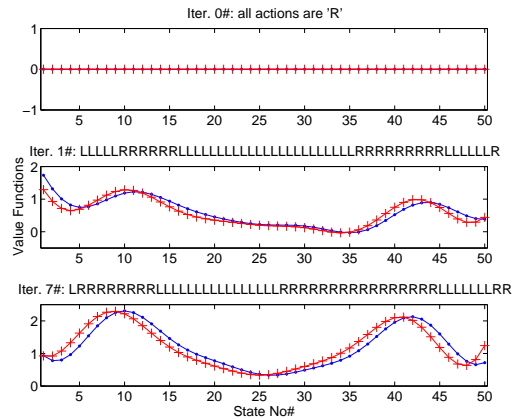

Figure 2: LSPI on the 50-state chain-walk, shown are iteration 0 (initialization) plus iteration 1 and 7.

initialized to the one-step approximated reward, [2] while the initial (state-action) value function for LSPI was 0.

The 50-state Chain. The problem is taken from [Lagoudakis and Parr, 2003] and it extends the 4-state chain in an obvious manner, except that the reward is also changed so that the reward is zero everywhere except at states 10 and 41, where it is one. We also used exactly the same radial basis features as [Lagoudakis and Parr, 2003], giving 22-dimensional state-action features (used by LSPI), and 11-dimensional state features (used by LAM-LSTD). We used a sample set comprising 100 pairs for each state-action pair. In total, LSPI spent 14 iterations to converge, and found the optimal policy at the last iteration. The initialization plus two early iterations are shown in Figure 2, three sample successive iterations in Figure 3. LAM-LSTD spent 4 iterations to converge, and found the optimal policy at iteration 2 as

---

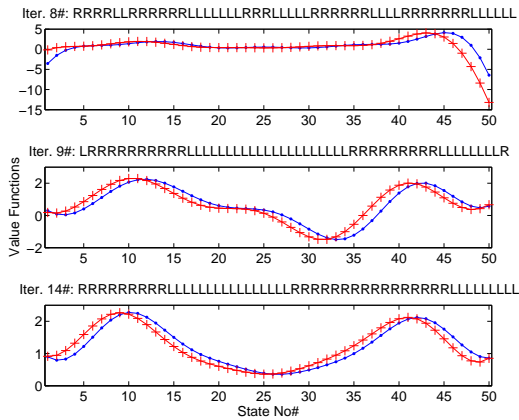[2] The initial state value function was indeed 0 for LAM-LSTD.

Figure 3: LSPI on the 50-state chain-walk *continued*: iteration 8, 9, 14. At iteration 14, LSPI converges.
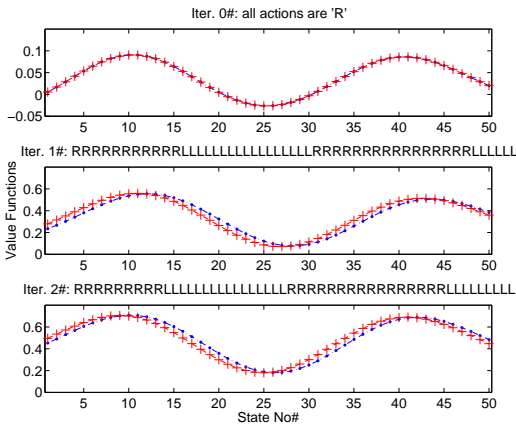

Figure 4: LAM-LSTD on the 50-state chain-walk. At iteration 2, the policy is already optimal. (LAM-LSTD converges in two more iterations.)

shown in Figure 4. After only one iteration, LAM-LSTD already found a suboptimal policy close the optimal one. In this experiment, a regularization factor of 100 was used for learning F and f. As the experiment with the 4-state chain, the weight vector was initialized to 0 for both algorithms. The first plot of LAM-LSTD, showing the approximated rewards by the features, indicates that the positions of the largest rewards are modeled correctly. With five polynomial bases, however, we found the positions of the largest reward are modeled incorrectly, and hence LAM-LSTD only gave a suboptimal policy, which is consistent to what has been described for LSPI by Lagoudakis and Parr [2003].

Inverted-pendulum

We used exactly the same simulator as Lagoudakis and Parr [2003]. The system is shown in Figure 5. The goal is to keep the pendulum above the horizontal line ($|\vartheta| \leq \pi/2$) for a
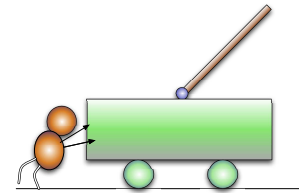

Figure 5: The pendulum task. One can choose between three actions at each time step: applying no force, pushing the cart to the right or left in 50 Newtons.

maximum of 3000 steps. A zero reward is given if the pendulum is above the horizontal line; otherwise a reward -1 is given at the same time of terminating the episode. The discount factor is 0.95. The state is composed of the angle and angular velocity of the pendulum. Both state variables are continuous.

The features of a state s are $\phi_i(s) = \exp(-\|s - u_{i-1}\|^2/2)$, $i = 1, 2, \ldots, 10$, where $u_0 = s$, and the other u_i are the points from the grid $\{-\pi/4, 0, \pi/4\} \times \{-1, 0, 1\}$. For further details on the problem, such as the dynamics of the system, we refer the reader to the original paper. The experiments used a maximum of 1000 episodes of data. The episodes were allowed for 3000 steps at maximum. In each episode, the pendulum was started from a uniformly random perturbation from the state $(0, 0)$, and the weight vector was initialized uniformly random. To study the data-efficiency of the methods, we gave the algorithms the first N epidoses, where $N \in \{50, 100, 150, \ldots, 1000\}$. After training on the N episodes we evaluated the learned policy and the number of balanced time steps were recorded. Thus, for each learning algorithm we got 20 measurements. In fact, this experiment was repeated 100 times to assess the stability of the algorithms.

The results are shown in Figures 6 and 7. In particular, Figure 6 shows the number of balancing steps, while Figure 7 shows the number of iterations of the algorithms, both as a function of the number of episodes used for training (i.e., N). Figure 6 shows that LAM-LSTD needs significantly fewer number of episodes than LSPI (in terms of both mean and variance of the balanced steps) to balance the same number of steps. At the same time, Figure 7 shows that LAM-LSTD requires much fewer iterations than LSPI. Note that the per iteration cost of LAM is approximately double the cost of LSPI, given that a matrix-vector operation is spent on the projection. Even with this increased cost, LAM-LSTD appears to be cheaper than LSPI. In generating the figures, a LAM-LSTD policy and a LSPI policy were learned from the same training data. No regulation was used for this experiment.

Conclusion

We have introduced a model-based API framework that uses linear action models. The linear action models are learned from data and then they are used in an approximate policy iteration method. Our proposed method is careful in using
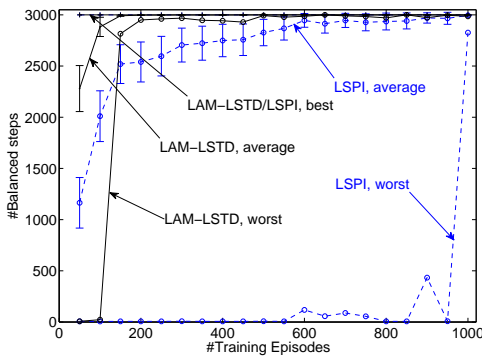
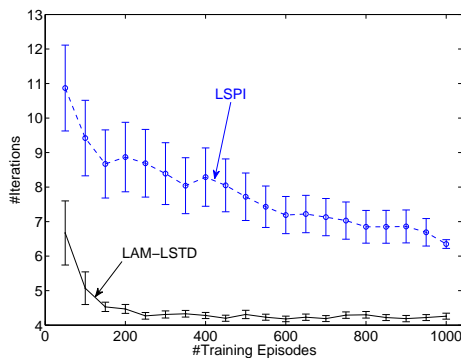Figure 6: Pendulum: balancing steps of LAM-LSTD and LSPI.

Figure 7: Pendulum: number of iterations of LAM-LSTD and LSPI.

the linear action models, in particular only on the features that occur in the data set. This is intended to ensure that the model predictions are accurate. As opposed to model-free methods, our approximate policy iteration method can use state value functions since the model can be used to predict action values. This might reduce the variance of the value-function estimates. In this paper we explored least-squares methods in both modeling and learning. The ideas in the paper extend readily to models of options, though this was not explored in the present paper.

A number of important issues remain open: First, it would be important to attempt a theoretical analysis of LAM-API. It may also be interesting to consider other model-building methods, e.g., different regularization terms, the "kernelization" of the model, or even considering non-linear action-models. It is trivial to extend our API method to consider features-vectors other than those available in the data. Although in this paper we argued heuristically against this idea, whether this is indeed a bad idea remains to be seen. Actually, a better question is how to generate additional feature-vectors to improve performance. Although our experimental results seem to suggest that LAM-API is a better algorithm than LSPI, further experimentation (and theoretical work) is needed to test this hypothesis.

References

D. P. Bertsekas and J. N. Tsitsiklis. *Neuro-dynamic Programming*. Athena, 1996.

J. A. Boyan. Technical update: Least-squares temporal difference learning. *Machine Learning*, 49:233–246, 2002.

S. Bradtke and A. G. Barto. Linear least-squares algorithms for temporal difference learning. *Machine Learning*, 22:33–57, 1996.

L. Buşoniu, A. Lazaric, M. Ghavamzadeh, R. Munos, R. Babuška, and B. Schutter. Least-squares methods for policy iteration. In M. Wiering and M. Otterlo, editors, *Reinforcement Learning*, volume 12 of *Adaptation, Learning, and Optimization*, pages 75–109. Springer Berlin Heidelberg, 2012.

R. A. Howard. *Dynamic Programming and Markov Processes*. The M.I.T. Press, 1960.

J-F. Hren and R. Munos. Optimistic planning of deterministic systems. In European Workshop on Reinforcement Learning Springer LNAI 5323, editor, *Recent Advances in Reinforcement Learning*, pages 151–164, 2008.

A. Isaza, Cs. Szepesvári, V. Bulitko, and R. Greiner. Speeding up planning in Markov decision processes via automatically constructed abstractions. In *UAI*, pages 306–314, 2008.

M. Lagoudakis and R. Parr. Least-squares policy iteration. *JMLR*, 4:1107–1149, 2003.

R. Parr, L. Li, G. Taylor, C. Painter-Wakefield, and M. L. Littman. An analysis of linear models, linear value-function approximation and feature selection for reinforcement learning. *ICML*, 2008.

J. Sorg and S. Singh. Linear options. In *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems (AAMAS '10)*, volume 1, pages 31–38, 2010.

R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1998.

R. S. Sutton, Cs. Szepesvári, A. Geramifard, and M. Bowling. Dyna-style planning with linear function approximation and prioritized sweeping. *UAI*, 2008.

R. S. Sutton, H. R. Maei, D. Precup, S. Bhatnagar, D. Silver, Cs. Szepesvári, and E. Wiewiora. Fast gradient-descent methods for temporal-difference learning with linear function approximation. *ICML*, 2009a.

R. S. Sutton, Cs. Szepesvári, and H. R. Maei. A convergent O(n) algorithm for off-policy temporal-difference learning with linear function approximation. *NIPS*, 2009b.

Cs. Szepesvári. *Algorithms for Reinforcement Learning*. Morgan and Claypool, July 2010.

M. Wiering and M. Otterlo, editors. *Reinforcement Learning*, volume 12 of *Adaptation, Learning, and Optimization*. Springer Berlin Heidelberg, 2012.

Y. Ye. The simplex method is strongly polynomial for the markov decision problem with a xed discount rate, 2010. URL http://www.stanford.edu/~yyye/simplexmdp1.pdf.