Least-Squares Temporal Difference Learning

Justin A. Boyan
NASA Ames Research Center
Moffett Field, CA 94035
jboyan@arc.nasa.gov

Abstract

TD(λ) is a popular family of algorithms for approximate policy evaluation in large MDPs. TD(λ) works by incrementally updating the value function after each observed transition. It has two major drawbacks: it makes inefficient use of data, and it requires the user to manually tune a stepsize schedule for good performance. For the case of linear value function approximations and $\lambda = 0$, the Least-Squares TD (LSTD) algorithm of Bradtke and Barto (Bradtke and Barto, 1996) eliminates all stepsize parameters and improves data efficiency.

This paper extends Bradtke and Barto's work in three significant ways. First, it presents a simpler derivation of the LSTD algorithm. Second, it generalizes from $\lambda = 0$ to arbitrary values of λ; at the extreme of $\lambda = 1$, the resulting algorithm is shown to be a practical formulation of supervised linear regression. Third, it presents a novel, intuitive interpretation of LSTD as a *model-based* reinforcement learning technique.

1 BACKGROUND

This paper addresses the problem of approximating the value function V^π of a fixed policy π in a large Markov decision process (Bertsekas and Tsitsiklis, 1996; Sutton and Barto, 1998). This is an important subproblem of several algorithms for sequential decision making, including optimistic policy iteration (Bertsekas and Tsitsiklis, 1996) and STAGE (Boyan and Moore, 1998). $V^\pi(x)$ simply predicts the expected long-term sum of future rewards obtained when the process starts in state x and follows policy π until termination. This function is well-defined as long as π is proper, i.e., guaranteed to terminate.[1]

For small Markov chains whose transition probabilities are all explicitly known, computing V^π is a trivial matter of solving a system of linear equations. However, in many practical applications, the transition probabilities of the chain are available only implicitly—either in the form of a *simulation model* or in the form of an agent's actual experience executing π in its environment. In either case, we must compute V^π or an approximation thereof (denoted $\tilde{V}^\pi$) solely from a collection of trajectories sampled from the chain. This is where the TD(λ) family of algorithms applies.

TD(λ) was introduced in (Sutton, 1988); excellent summaries may now be found in several books (Bertsekas and Tsitsiklis, 1996; Sutton and Barto, 1998). For each state on each observed trajectory, TD(λ) incrementally adjusts the coefficients of $\tilde{V}^\pi$ toward new target values. The target values depend on the parameter $\lambda \in [0, 1]$. At $\lambda = 1$, the target at each visited state x_t is the "Monte-Carlo return," i.e., the actual observed sum of future rewards $R_t + R_{t+1} + \cdots + R_{\text{END}}$. This is an unbiased sample of $V^\pi(x_t)$, but may have significant variance since it depends on a long stochastic sequence of rewards. At the other extreme, $\lambda = 0$, the target value is set by a sampled one-step lookahead: $R_t + \tilde{V}^\pi(x_{t+1})$. This value has lower variance—the only random component is a single state transition—but is biased by the potential inaccuracy of the lookahead estimate of V^π. The parameter λ trades off between bias and variance. Empirically, intermediate values of λ seem to perform best (Sutton, 1988;

[1] For improper policies, V^π may be made well-defined by the use of a discount factor that exponentially reduces future rewards; however, for simplicity we will assume here that V^π is undiscounted.

Sutton and Barto, 1998).

TD(λ) has been shown to converge to a good approximation of V^{π} when *linear architectures* are used, assuming a suitable decreasing schedule of stepsizes for the incremental weight updates (Tsitsiklis and Roy, 1996). Linear architectures—which include lookup tables, state aggregation methods, CMACs, radial basis function networks with fixed bases, and multi-dimensional polynomial regression—approximate $V^{\pi}(x)$ by first mapping the state x to a feature vector $\boldsymbol{\phi}(x) \in \Re^K$, and then computing a linear combination of those features, $\boldsymbol{\phi}(x)^{\mathsf{T}}\boldsymbol{\beta}$. Figure 1 gives a convenient form of TD(λ) that exploits this representation.

On each transition, the algorithm computes the scalar one-step TD error $R_t + (\boldsymbol{\phi}(x_{t+1}) - \boldsymbol{\phi}(x_t))^{\mathsf{T}}\boldsymbol{\beta}$, and apportions that error among all state features according to their respective *eligibilities* $\mathbf{z}_t$. The eligibility vector may be seen as an algebraic trick by which TD(λ) propagates rewards backward over the current trajectory without having to remember the trajectory explicitly. Each feature's eligibility at time t depends on the trajectory's history and on λ: $\mathbf{z}_t = \sum_{i=t_0}^{t} \lambda^{t-i}\boldsymbol{\phi}(x_i)$, where t_0 is the time at which the current trajectory started. In the case of TD(0), only the current state's features are eligible to be updated, so $\mathbf{z}_t = \boldsymbol{\phi}(x_t)$; whereas in TD(1), the features of all states seen so far on the current trajectory are eligible, so $\mathbf{z}_t = \sum_{i=t_0}^{t} \boldsymbol{\phi}(x_i)$.

To what weights does TD(λ) converge? Examining the update rule for $\boldsymbol{\delta}$ in Figure 1, it is not difficult to see that the coefficient changes made by TD(λ) after an observed trajectory $(x_0, x_1, \ldots, x_L, \text{END})$ have the form $\boldsymbol{\beta} := \boldsymbol{\beta} + \alpha_n(\mathbf{d} + \mathbf{C}\boldsymbol{\beta} + \boldsymbol{\omega})$, where

$$\mathbf{d} = \mathrm{E}\Big\{\sum_{i=0}^{L} \mathbf{z}_i R_i\Big\}; \quad \mathbf{C} = \mathrm{E}\Big\{\sum_{i=0}^{L} \mathbf{z}_i\big(\boldsymbol{\phi}(x_{i+1}) - \boldsymbol{\phi}(x_i)\big)^{\mathsf{T}}\Big\}; \tag{1}$$

and $\boldsymbol{\omega}$ = zero-mean noise. The expectations are taken with respect to the distribution of trajectories through the Markov chain. It is shown in (Bertsekas and Tsitsiklis, 1996) that $\mathbf{C}$ is negative definite and that the noise $\boldsymbol{\omega}$ has sufficiently small variance, which together with the stepsize conditions mentioned above, imply that $\boldsymbol{\beta}$ converges to a fixed point $\boldsymbol{\beta}_\lambda$ satisfying $\mathbf{d} + \mathbf{C}\boldsymbol{\beta}_\lambda = \mathbf{0}$. In effect, TD($\lambda$) solves this system of equations by performing stochastic gradient descent on a potential function $\|\boldsymbol{\beta} - \boldsymbol{\beta}_\lambda\|^2$. It never explicitly represents $\mathbf{d}$ or $\mathbf{C}$. The changes to $\boldsymbol{\beta}$ depend only on the most recent trajectory, and after those changes are made, the trajectory and its rewards are simply forgotten. This approach, while requiring little computation per iteration, wastes data and may require sampling many trajectories to reach convergence.

One technique for using data more efficiently is "experience replay" (Lin, 1993): explicitly remember all trajectories ever seen, and whenever asked to produce an updated set of coefficients, perform repeated passes of TD(λ) over all the saved trajectories until convergence. This technique is similar to the batch training methods commonly used to train neural networks. However, in the case of linear function approximators, there is another way.

2 THE LEAST-SQUARES TD($\boldsymbol{\lambda}$) ALGORITHM

The Least-Squares TD(λ) algorithm, or LSTD(λ), converges to the same coefficients $\boldsymbol{\beta}_\lambda$ that TD(λ) does. However, instead of performing gradient descent, LSTD(λ) builds explicit estimates of the $\mathbf{C}$ matrix and $\mathbf{d}$ vector (actually, estimates of a constant multiple of $\mathbf{C}$ and $\mathbf{d}$), and then solves $\mathbf{d} + \mathbf{C}\boldsymbol{\beta}_\lambda = \mathbf{0}$ directly. The actual data structures that LSTD(λ) builds from experience are the matrix $\mathbf{A}$ (of dimension $K \times K$, where K is the number of features) and the vector $\mathbf{b}$ (of dimension K):

$$\mathbf{b} = \sum_{i=0}^{t} \mathbf{z}_i R_i \quad \mathbf{A} = \sum_{i=0}^{t} \mathbf{z}_i\big(\boldsymbol{\phi}(x_i) - \boldsymbol{\phi}(x_{i+1})\big)^{\mathsf{T}} \tag{2}$$

After n independent trajectories have been observed, $\mathbf{b}$ is an unbiased estimate of $n\mathbf{d}$, and $\mathbf{A}$ is an unbiased estimate of $-n\mathbf{C}$. Thus, $\boldsymbol{\beta}_\lambda$ can be estimated as $\mathbf{A}^{-1}\mathbf{b}$. As is standard in least-squares algorithms, Singular Value Decomposition is used to invert $\mathbf{A}$ robustly (Press *et al.*, 1992). The complete LSTD(λ) algorithm is specified in Figure 2.

When $\lambda = 0$, LSTD(0) reduces precisely to Bradtke and Barto's LSTD algorithm, which they derived using a more complex approach based on regression with instrumental variables (Bradtke and Barto, 1996). At the other extreme, when $\lambda = 1$, LSTD(1) produces the same $\mathbf{A}$ and $\mathbf{b}$ that would be produced by supervised linear regression on training pairs of {state features $\mapsto$ observed Monte-Carlo returns} (see (Boyan, 1998) for proof). Thanks to the algebraic trick of the eligibility vectors, LSTD(1) builds the regression matrices *fully incrementally*—without having to store the trajectory while waiting to observe the eventual outcome. When trajectories through the chain are long, this provides significant memory savings over linear regression.

TD($\boldsymbol{\lambda}$) for approximate policy evaluation:

Given: • a *simulation model* for a proper policy π in MDP X;
• a *featurizer* $\boldsymbol{\phi} : X \to \Re^K$ mapping states to feature vectors, $\boldsymbol{\phi}(\text{END}) \stackrel{\text{def}}{=} \mathbf{0}$;
• a parameter $\lambda \in [0, 1]$; and
• a sequence of *stepsizes* $\alpha_1, \alpha_2, \ldots$ for incremental coefficient updating.

Output: a coefficient vector $\boldsymbol{\beta}$ for which $V^\pi(x) \approx \boldsymbol{\beta} \cdot \boldsymbol{\phi}(x)$.

Set $\boldsymbol{\beta} := \mathbf{0}$ *(or an arbitrary initial estimate)*, $t := 0$.
for $n := 1, 2, \ldots$ **do**: {
  Set $\boldsymbol{\delta} := 0$.
  Choose a start state $x_t \in X$.
  Set $\mathbf{z}_t := \boldsymbol{\phi}(x_t)$.
  while $x_t \neq \text{END}$, **do**: {
    Simulate one step of the process, producing a reward R_t and next state x_{t+1}.
    Set $\boldsymbol{\delta} := \boldsymbol{\delta} + \mathbf{z}_t\big(R_t + (\boldsymbol{\phi}(x_{t+1}) - \boldsymbol{\phi}(x_t))^\mathsf{T}\boldsymbol{\beta}\big)$. /* *inner product* */
    Set $\mathbf{z}_{t+1} := \lambda\mathbf{z}_t + \boldsymbol{\phi}(x_{t+1})$.
    Set $t := t + 1$.
  }
  Set $\boldsymbol{\beta} := \boldsymbol{\beta} + \alpha_n\boldsymbol{\delta}$.
}

Figure 1: Ordinary TD(λ) for linearly approximating the undiscounted value function of a fixed proper policy.

LSTD($\boldsymbol{\lambda}$) for approximate policy evaluation:

Given: a *simulation model*, *featurizer*, and λ as in ordinary TD(λ).
(No stepsize schedules or initial estimates of $\boldsymbol{\beta}$ are necessary.)

Output: a coefficient vector $\boldsymbol{\beta}$ for which $V^\pi(x) \approx \boldsymbol{\beta} \cdot \boldsymbol{\phi}(x)$.

Set $\mathbf{A} := \mathbf{0}$, $\mathbf{b} := \mathbf{0}$, $t := 0$.
for $n := 1, 2, \ldots$ **do**: {
  Choose a start state $x_t \in X$.
  Set $\mathbf{z}_t := \boldsymbol{\phi}(x_t)$.
  while $x_t \neq \text{END}$, **do**: {
    Simulate one step of the chain, producing a reward R_t and next state x_{t+1}.
    Set $\mathbf{A} := \mathbf{A} + \mathbf{z}_t(\boldsymbol{\phi}(x_t) - \boldsymbol{\phi}(x_{t+1}))^\mathsf{T}$. /* *outer product* */
    Set $\mathbf{b} := \mathbf{b} + \mathbf{z}_t R_t$.
    Set $\mathbf{z}_{t+1} := \lambda\mathbf{z}_t + \boldsymbol{\phi}(x_{t+1})$.
    Set $t := t + 1$.
  }
  Whenever updated coefficients are desired: Set $\boldsymbol{\beta} := \mathbf{A}^{-1}\mathbf{b}$. /* *use Singular Value Decomposition* */
}

Figure 2: A least-squares version of TD(λ) (compare Figure 1). Note that $\mathbf{A}$ has dimension $K \times K$, and $\mathbf{b}$, $\boldsymbol{\beta}$, $\mathbf{z}$, and $\boldsymbol{\phi}(x)$ all have dimension $K \times 1$.

The computation per timestep required to update $\mathbf{A}$ and $\mathbf{b}$ is the same as least-squares linear regression: $O(K^2)$, where K is the number of features. LSTD(λ) must also perform a matrix inversion at a cost of $O(K^3)$ whenever $\boldsymbol{\beta}$'s coefficients are needed—typically, once per complete trajectory. (If updated coefficients are required more frequently, then the $O(K^3)$ cost can be avoided by *recursive least-squares* (Bradtke and Barto, 1996) or Kalman-filtering techniques (Bertsekas and Tsitsiklis, 1996, §3.2.2), which update $\boldsymbol{\beta}$ on each timestep at a cost of only $O(K^2)$.) LSTD(λ) performs more computation per observation than incremental TD(λ), which updates the coefficients using only $O(K)$ computation per timestep. However, LSTD(λ) offers several significant advantages:

- Least-squares algorithms "extract more information from each additional observation" (Bradtke and Barto, 1996) and would thus be expected to converge with fewer training samples.
- TD(λ)'s convergence can be slowed dramatically by a poor choice of the stepsize parameters α_n. LSTD(λ) eliminates these parameters.
- TD(λ)'s performance is sensitive to the initial estimate for $\boldsymbol{\beta}_\lambda$. LSTD($\lambda$) does not rely on an arbitrary initial estimate.
- TD(λ) is also sensitive to the ranges of the individual features. LSTD(λ) is not.

Section 4 below presents experimental results comparing the data efficiency of gradient-based and least-squares-based TD learning.

3 LSTD(λ) AS MODEL-BASED LEARNING

Surprisingly, the move from a gradient-based to a least-squares-based update rule for TD(λ) turns out to be mathematically equivalent to a move from a model-free to a model-based reinforcement learning algorithm. This equivalence provides interesting new intuitions about the space of temporal difference learning algorithms, and as such forms an important part of this work's contribution.

To begin, let us restrict our attention to the case of a small discrete state space X, over which V^π can be represented and learned exactly by a lookup table. A classical model-based algorithm for learning V^π from simulated trajectory data would proceed as follows:

1. From the state transitions and rewards observed so far, build in memory an *empirical model* of the Markov chain. The sufficient statistics of this model are
   - a vector $\mathbf{n}$ recording the number of times each state has been visited;
   - a matrix $\mathbf{C}$ recording the observed state-transition counts: $\mathbf{C}_{ij}$ = how many times x_j was seen to directly follow x_i; and
   - a vector $\mathbf{s}$ recording, for each state, the sum of all one-step rewards observed on transitions leaving that state.
2. Whenever a new estimate of the value function V^π is desired, solve the linear system of Bellman equations corresponding to the current empirical model. Writing $\mathbf{N} = \mathbf{diag}(\mathbf{n})$, the solution vector of V^π values is given by

$$\mathbf{v} = (\mathbf{N} - \mathbf{C})^{-1}\mathbf{s}. \tag{3}$$

This model-based technique contrasts with TD(λ), a model-free approach to the same problem. TD(λ) does not maintain any statistics on observed transitions and rewards; it simply updates the components of $\mathbf{v}$ directly. In the limit, assuming a lookup-table representation, both converge to the optimal V^π. The advantage of TD(λ) is its low computational burden per step; the advantage of the classical model-based method is that it makes the most of the available training data. The empirical advantages of model-based and model-free reinforcement learning methods have been investigated in, e.g., (Sutton, 1990; Moore and Atkeson, 1993; Atkeson and Santamaria, 1997).

Where does LSTD(λ) fit in? In fact, for the case of $\lambda = 0$, it precisely duplicates the classical model-based method sketched above. The assumed lookup-table representation for $\tilde{V}^\pi$ means that we have one independent feature per state: the feature vector $\boldsymbol{\phi}$ corresponding to state 1 is $(1, 0, 0, \ldots, 0)$; corresponding to state 2 is $(0, 1, 0, \ldots, 0)$; etc. Referring to the algorithm of Figure 2, we see that LSTD(0) performs the following operations upon each observed transition:

$$\mathbf{b} := \mathbf{b} + \boldsymbol{\phi}(x_t)R_t \quad \mathbf{A} := \mathbf{A} + \boldsymbol{\phi}(x_t)(\boldsymbol{\phi}(x_t) - \boldsymbol{\phi}(x_{t+1}))^\mathsf{T} \tag{4}$$

Clearly, the role of $\mathbf{b}$ is to sum all the rewards observed at each state, exactly as the vector $\mathbf{s}$ does in the classical technique. $\mathbf{A}$, meanwhile, accumulates the statistics $(\mathbf{N} - \mathbf{C})$. To see this, note that the outer product in Eq. 4 is a matrix consisting of an entry of $+1$ on the single diagonal element corresponding to state x_t; an entry of -1 on the element in row x_t, column x_{t+1}; and all the rest zeroes. Summing one such sparse matrix for each observed transition gives $\mathbf{A} \equiv \mathbf{N} - \mathbf{C}$. Finally, LSTD(0) performs the inversion $\boldsymbol{\beta} := \mathbf{A}^{-1}\mathbf{b} = (\mathbf{N} - \mathbf{C})^{-1}\mathbf{s}$, giving the same solution as in Equation 3.

Thus, when $\lambda = 0$, the $\mathbf{A}$ and $\mathbf{b}$ matrices built by LSTD(λ) effectively record a model of all the observed transitions. What about when $\lambda > 0$? Again, $\mathbf{A}$ and $\mathbf{b}$ record the sufficient statistics of an empirical Markov model—but in this case, the model being captured is one whose single-step transition probabilities directly encode the multi-step TD(λ) backup operations. That is, the model links each state x to all the downstream states that follow x on any trajectory, and records how much influence each has on estimating $\tilde{V}^\pi(x)$ according to TD(λ). In the case of $\lambda = 0$, the TD(λ) backups correspond to the one-step transitions, resulting in the equivalence described above. The opposite extreme, the case of $\lambda = 1$, is also interesting: the empirical Markov model corresponding to TD(1)'s backups is the chain in which each state x leads directly to absorption, and $\boldsymbol{\beta}$ then simply computes the average Monte-Carlo return at each state. In short, if we assume a lookup-table representation for the function $\tilde{V}^\pi$, we can view the LSTD(λ) algorithm as performing these two steps:

1. It implicitly uses the observed simulation data to build a Markov chain. This chain compactly models all the backups that TD(λ) would perform on the data.
2. It solves the chain by performing a matrix inversion.

The lookup-table representation for $\tilde{V}^\pi$ is intractable in practical problems; in practice, LSTD(λ) operates on states only via their (linearly dependent) feature representations $\boldsymbol{\phi}(x)$. In this case, we can view LSTD(λ) as implicitly building a *compressed* version of the empirical model's transition matrix $\mathbf{N} - \mathbf{C}$ and summed-reward vector $\mathbf{s}$:

$$\mathbf{b} = \boldsymbol{\Phi}^{\mathsf{T}}\mathbf{s} \qquad\qquad \mathbf{A} = \boldsymbol{\Phi}^{\mathsf{T}}(\mathbf{N} - \mathbf{C})\boldsymbol{\Phi} \tag{5}$$

where $\boldsymbol{\Phi}$ is the $|X| \times K$ matrix representation of the function $\boldsymbol{\phi} : X \to \Re^K$. From the compressed empirical model, LSTD(λ) computes the following coefficients for $\tilde{V}^\pi$:

$$\boldsymbol{\beta}_\lambda = \mathbf{A}^{-1}\mathbf{b} = (\boldsymbol{\Phi}^{\mathsf{T}}(\mathbf{N} - \mathbf{C})\boldsymbol{\Phi})^{-1}(\boldsymbol{\Phi}^{\mathsf{T}}\mathbf{s}). \tag{6}$$

Ideally, these coefficients $\boldsymbol{\beta}_\lambda$ would be equivalent to the *empirical optimal* coefficients $\boldsymbol{\beta}^*_\lambda$. The empirical optimal coefficients are those that would be found by building the full uncompressed empirical model (represented by $\mathbf{N} - \mathbf{C}$ and $\mathbf{s}$), using a lookup table to solve for that model's value function ($\mathbf{v} = (\mathbf{N} - \mathbf{C})^{-1}\mathbf{s}$), and then performing a least-squares linear fit from the state features $\boldsymbol{\Phi}$ to the lookup-table value function:

$$\boldsymbol{\beta}^*_\lambda \stackrel{\text{def}}{=} (\boldsymbol{\Phi}^{\mathsf{T}}\boldsymbol{\Phi})^{-1}(\boldsymbol{\Phi}^{\mathsf{T}}\mathbf{v}) = (\boldsymbol{\Phi}^{\mathsf{T}}\boldsymbol{\Phi})^{-1}\boldsymbol{\Phi}^{\mathsf{T}}(\mathbf{N} - \mathbf{C})^{-1}\mathbf{s}. \tag{7}$$

It can be shown that Equations 6 and 7 are indeed equivalent for the case of $\lambda = 1$, because that setting of λ implies that $\mathbf{C} = \mathbf{0}$ (thus $(\mathbf{N} - \mathbf{C})^{-1}$ is diagonal and commutes). However, for the case of $\lambda < 1$, solving the compressed empirical model does not in general produce the optimal least-squares fit to the solution of the uncompressed model.

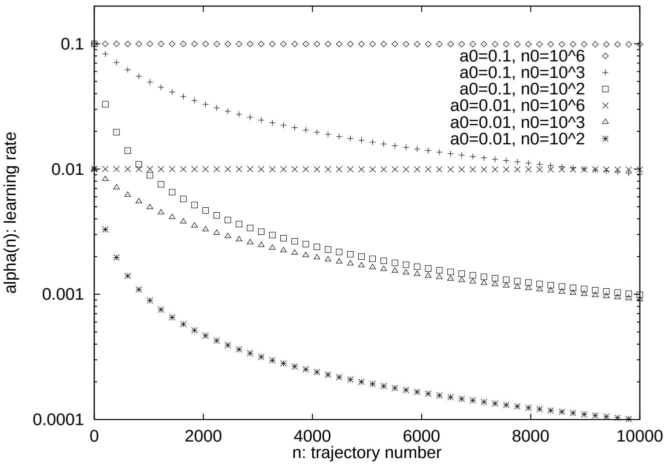

Figure 3: The six different *stepsize schedules* used in the experiments with TD(λ). The schedules are determined by Equation 8 with various settings for a_0 and n_0.

4 EXPERIMENTAL COMPARISON OF TD($\boldsymbol{\lambda}$) AND LSTD($\boldsymbol{\lambda}$)

This section reports experimental results comparing TD(λ) and LSTD(λ) on the simple Markov chain illustrated in Figure 4. The chain consists of 13 states, and we seek to represent its value function compactly as a linear function of four state features as shown. This domain's optimal V^π function is exactly linear in these features: the optimal coefficients $\boldsymbol{\beta}^*_\lambda$ are $(-24, -16, -8, 0)$. This condition guarantees

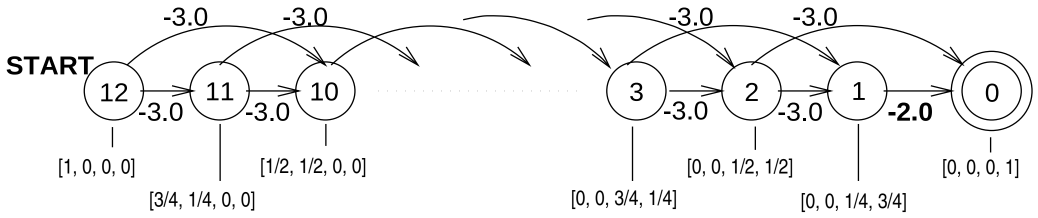

Figure 4: A 13-state Markov chain. Each state is represented by four features as shown. In states 2–12, each outgoing arc is taken with probability 0.5.

that LSTD(λ) will converge with probability 1 to the optimal $\boldsymbol{\beta}^*_\lambda$ for any setting of λ.

TD(λ) is also guaranteed to converge to the optimal V^π, under the additional condition that an appropriate schedule of stepsizes is chosen. The following three criteria on the schedule (α_n) are sufficient: $\alpha_n \geq 0 \ \ \forall n$; $\sum_{n=1}^\infty \alpha_n = \infty$; and $\sum_{n=1}^\infty \alpha_n^2 < \infty$. Our experiments use schedules that satisfy these criteria, having the following form:

$$\alpha_n \stackrel{\text{def}}{=} a_0 \frac{n_0+1}{n_0+n} \qquad n = 1, 2, \ldots \tag{8}$$

The parameter a_0 determines the initial stepsize, and n_0 determines how gradually the stepsize decreases over time. Each TD(λ) experiment was run with six different stepsize schedules, corresponding to the six combinations of $a_0 \in \{0.1, 0.01\}$ and $n_0 \in \{10^2, 10^3, 10^6\}$. These six schedules are plotted in Figure 3, from which it can be seen that they fall into a typical range of learning rates used in applications of gradient descent.

Comparative results are given in Figures 5 and 6. Figure 5 focuses on the case of $\lambda = 0.4$, comparing the learning curve for LSTD(λ) against those of all six schedules of TD(λ). Each point plotted represents the average of 10 trials. The plot shows clearly that for $\lambda = 0.4$, LSTD(λ) learns a good approximation to V^π in fewer trials than any of the TD(λ) experiments, and performs better asymptotically as well.

Figure 6 graphically summarizes six learning-curve plots similar to Figure 5, corresponding to varying λ over the range $\{0, 0.2, 0.4, 0.6, 0.8, 1.0\}$. The results may be summarized as follows:

- Across all values of λ, LSTD(λ) learns a good approximation to V^π in fewer trials than any of the TD(λ) experiments, and performs better asymptotically as well.
- The performance of TD(λ) depends critically on the stepsize schedule chosen. LSTD(λ) has no tunable parameters other than λ itself.
- Varying λ has a relatively small effect on LSTD(λ)'s performance.

5 CONCLUSIONS AND FUTURE WORK

We have argued, both from the experimental results above and from the deep connection to model-based reinforcement learning presented in Section 3, that the least-squares formulation of TD learning makes better use of simulation data than TD(λ). Further experiments are needed to determine when to prefer one algorithm over the other as a practical matter. If a domain has many features and simulation data is available cheaply, then incremental methods such as TD(λ) may have better real-time performance than least-squares methods (Sutton, 1992). On the other hand, some reinforcement learning applications have been successful with very small numbers of features (e.g., (Singh and Bertsekas, 1997; Boyan and Moore, 1998)), and in these situations LSTD(λ) should be superior.

LSTD(λ) has been successfully applied in the context of STAGE, a reinforcement learning algorithm for combinatorial optimization (Boyan, 1998). An exciting possibility for future work is to apply LSTD(λ) in the context of approximation algorithms for general Markov decision problems. LSTD(λ) provides an alternative to TD(λ) for the inner loop of optimistic policy iteration (Bertsekas and Tsitsiklis, 1996), and should thereby enable good control policies to be discovered with fewer trial simulations.

Acknowledgments

Thanks to Andrew Moore and Jeff Schneider for helpful comments. This work was supported by a NASA GSRP fellowship while the author was at CMU.

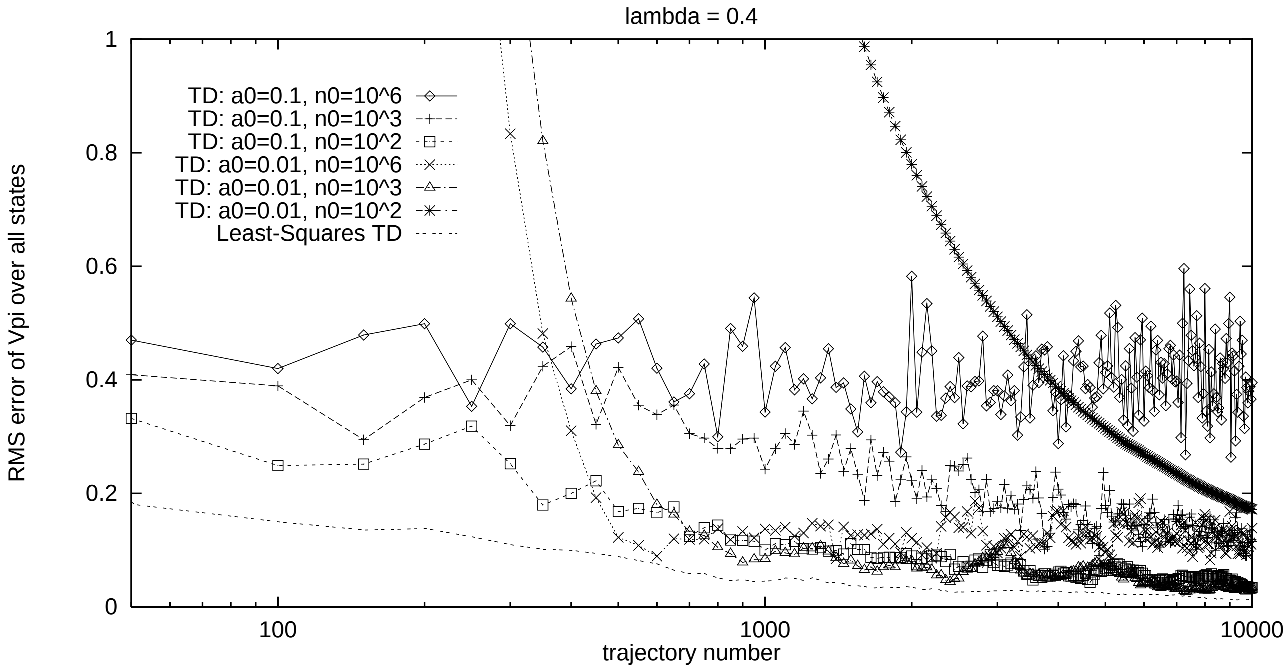

Figure 5: Performance of TD(0.4) and LSTD(0.4) on the sample domain. Note the log scale on the x-axis. All points plotted represent the average of 10 trials.

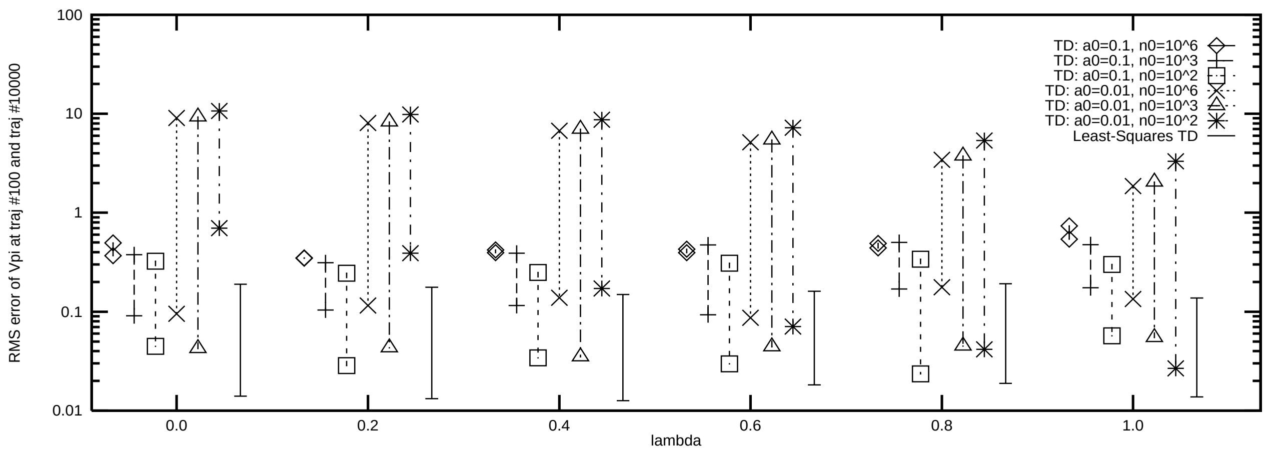

Figure 6: Summary of results at six settings of λ. At each setting, seven algorithms are compared: TD(λ) (with six different stepsize schedules) and LSTD(λ). The plotted segment shows the mean RMS value function approximation error after 100 trajectories (top of segment) and 10,000 trajectories (bottom of segment). Note the log scale on the y-axis. LSTD(λ) is best in all cases.

References

[Atkeson and Santamaria, 1997] C. G. Atkeson and J. C. Santamaria. A comparison of direct and model-based reinforcement learning. In *International Conference on Robotics and Automation*, 1997.

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

[Boyan and Moore, 1998] J. A. Boyan and A. W. Moore. Learning evaluation functions for global optimization and Boolean satisfiability. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI)*, 1998.

[Boyan, 1998] J. A. Boyan. *Learning Evaluation Functions for Global Optimization.* PhD thesis, Carnegie Mellon University, 1998.

[Bradtke and Barto, 1996] S. J. Bradtke and A. G. Barto. Linear least-squares algorithms for temporal difference learning. *Machine Learning*, 22(1/2/3):33–57, 1996.

[Lin, 1993] L.-J. Lin. *Reinforcement Learning for Robots Using Neural Networks.* PhD thesis, Carnegie Mellon University, 1993.

[Moore and Atkeson, 1993] A. W. Moore and C. G. Atkeson. Prioritized sweeping: Reinforcement learning with less data and less time. *Machine Learning*, 13:103–130, 1993.

[Press *et al.*, 1992] W.H. Press, S.A. Teukolsky, W.T. Vetterling, and B.P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing.* Cambridge University Press, second edition, 1992.

[Singh and Bertsekas, 1997] S. Singh and D. Bertsekas. Reinforcement learning for dynamic channel allocation in cellular telephone systems. In M. C. Mozer, M. I. Jordan, and T. Petsche, editors, *NIPS-9*, page 974. The MIT Press, 1997.

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

[Sutton, 1988] R. S. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3, 1988.

[Sutton, 1990] R. S. Sutton. Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *Proceedings of the Seventh International Conference on Machine Learning.* Morgan Kaufmann, 1990.

[Sutton, 1992] R. S. Sutton. Gain adaptation beats least squares. In *Proceedings of the* 7^{th} *Yale Workshop on Adaptive and Learning Systems*, pages 161–166, 1992.

[Tsitsiklis and Roy, 1996] J. N. Tsitsiklis and B. Van Roy. An analysis of temporal-difference learning with function approximation. Technical Report LIDS-P-2322, MIT, 1996.