Incremental Least-Squares Temporal Difference Learning

Alborz Geramifard   Michael Bowling   Richard S. Sutton

Department of Computing Science
University of Alberta
Edmonton, Alberta, T6G 2E8, Canada
{alborz, bowling, sutton}@cs.ualberta.ca

Abstract

Approximate policy evaluation with linear function approximation is a commonly arising problem in reinforcement learning, usually solved using temporal difference (TD) algorithms. In this paper we introduce a new variant of linear TD learning, called *incremental least-squares TD learning*, or iLSTD. This method is more data efficient than conventional TD algorithms such as TD(0) and is more computationally efficient than non-incremental least-squares TD methods such as LSTD (Bradtke & Barto 1996; Boyan 1999). In particular, we show that the per-time-step complexities of iLSTD and TD(0) are $O(n)$, where n is the number of features, whereas that of LSTD is $O(n^2)$. This difference can be decisive in modern applications of reinforcement learning where the use of a large number features has proven to be an effective solution strategy. We present empirical comparisons, using the test problem introduced by Boyan (1999), in which iLSTD converges faster than TD(0) and almost as fast as LSTD.

Introduction

Policy evaluation strives to approximate the value function of a fixed policy in a Markov decision process and is a key component of the family of reinforcement-learning algorithms based on policy iteration. This paper specifically focuses on the problem of *online policy evaluation*, where an approximate value function is maintained and updated after each time step of following the policy. In particular we examine the case of linear value function approximation, which has been the focus of recent theoretical and practical research.

Temporal difference (TD) learning (Sutton 1988) is the traditional approach to policy evaluation in reinforcement learning. TD learning can be guaranteed to converge with any linear function approximator and appropriate step-size schedule (Tsitsiklis & Van Roy 1997). In addition, TD learning is computationally inexpensive, requiring only $O(n)$ computation per time step where n is the number of state features used in the approximator. However, conventional TD methods do not use trajectory data optimally. After performing a gradient update, the state transitions and rewards are simply forgotten. Experience replay (Lin 1993) is an approach to improve the data efficiency of TD learning by

saving trajectories and repeatedly performing gradient updates over the saved trajectories. In this paper we focus on TD(0), the one-step TD algorithm for policy evaluation.

The least-squares TD algorithm (LSTD) is a recent alternative proposed by Bradtke and Barto (1996) and extended by Boyan (1999; 2002) and Xu *et al.* (2002). LSTD explicitly solves for the value function parameters that result in zero mean TD update over all observed state transitions. The resulting algorithm is considerably more data efficient than TD(0), but less computationally efficient. Even using incremental inverse computations LSTD still requires $O(n^2)$ computation per time step. Hence, the improved data efficiency is obtained at a substantial computational price.

The tradeoff between data and computational efficiency is felt most when the value function approximator involves a very large number of features. A large number of features, though, is not a rare special case, but rather the norm in applications of reinforcement learning with linear function approximators (*e.g.*, Sutton 1996; Stone, Sutton, & Kuhlmann 2005; Tedrake, Zhang, & Seung 2004). Large feature sets arise naturally in large problems, where they are used to avoid the theoretical quagmire of general non-linear function approximation. Tile coding (*e.g.*, Albus 1971), coarse coding (*e.g.*, Hinton, McClelland, & Rumelhart 1986), and radial basis functions (*e.g.*, Poggio & Girosi 1990) all construct non-linear value functions by learning linear functions in a modified and very large feature representation. The number of features considered can be in the millions or more, reaching the point where the quadratic cost of LSTD is impractical.

One typical property of high dimensional feature representations, such as tile coding, is that often only a small number of features are "on" (*i.e.*, non-zero) for any given state. In this paper we focus on this common case, proposing a compromise between the computational efficiency of TD(0) and the data efficiency of LSTD. We introduce iLSTD, which like LSTD seeks to minimize the mean TD update over all of the observed trajectories. However, when only a small number of features are non-zero, iLSTD requires only $O(n)$ computation per time step.

We begin by presenting an overview of the TD(0), LSTD, and their computational demands. We follow with a description of iLSTD, showing that its computational complexity per time step is indeed linear in the number of features.

We present empirical results comparing TD(0), LSTD, and iLSTD, showing that iLSTD obtains the key advantages of both of the other two methods: it is nearly as data efficient as LSTD and nearly as computationally efficient as TD(0). We discuss some future directions before concluding.

Background

Reinforcement learning is an approach to sequential decision making in an unknown environment by learning from past interactions with that environment (*e.g.*, see Sutton & Barto 1998). This paper specifically considers the class of environments known as Markov decision processes (MDPs). An MDP is a tuple, $(\mathcal{S}, \mathcal{A}, \mathcal{P}^a_{ss'}, \mathcal{R}^a_{ss'}, \gamma)$, where $\mathcal{S}$ is a set of states, $\mathcal{A}$ is a set of actions, $\mathcal{P}^a_{ss'}$ is the probability of reaching state s' after taking action a in state s, and $\mathcal{R}^a_{ss'}$ is the reward received when that transition occurs, and $\gamma \in [0, 1]$ is a discount rate parameter. A trajectory of experience is a sequence $s_1, a_1, r_2, s_2, a_2, r_3, s_3, \ldots$, where the agent in s_1 takes action a_1 and receives reward r_2 while transitioning to s_2 before taking a_2, *etc.*

In this work we focus on the problem of learning an approximation of a policy's state-value function from sample trajectories of experience following that policy. A method for solving this problem is a key component of many reinforcement learning algorithms. In particular, maintaining an online estimate of the value function can be combined with policy improvement to learn a controller. The value of a state given a policy is the expected sum of discounted future rewards:

$$V^\pi(s) = E\left[\sum_{t=1}^{\infty} \gamma^{t-1} r_t \,\middle|\, s_0 = s, \pi\right].$$

We can write the value function recursively as

$$
\begin{aligned}
V^\pi(s) &= \sum_a \pi(s, a) \sum_{s'} \mathcal{P}^a_{ss'} \left[\mathcal{R}^a_{ss'} + \gamma V^\pi(s')\right] \\
&= E\left[r_{t+1} + \gamma V^\pi(s_{t+1}) | s_t = s, \pi\right]. \quad (1)
\end{aligned}
$$

The recursive relationship involves an implicit expectation which is made explicit in Equation 1. Notice that trajectories of experience can be seen as samples of this expectation. For a particular value function V let the TD error at time t be defined as,

$$\delta_t(V) = r_{t+1} + \gamma V(s_{t+1}) - V(s_t). \quad (2)$$

Then, $E_t[\delta_t(V^\pi)] = 0$, that is, the mean TD error for the policy's true value function must be zero.

We are interested in approximating V^π using a linear function approximator. In particular, suppose we have a function $\phi : \mathcal{S} \to \Re^n$, which gives a feature representation of the state space. We are interested in approximate value functions of the form $V_\theta(s) = \phi(s)^T \theta$, where $\theta \in \Re^n$ are the parameters of the value function. The focus of this work is on situations where the feature representation is sparse, *i.e.*, for all states s the number of non-zero features in $\phi(s)$ is no more than $k \ll n$. This situation arises often when using feature representations such as tile-coding. For example, in Stone, Sutton, and Kuhlmann's (2005) work on learning

0	$s \leftarrow s_0$
1	Initialize θ arbitrarily
2	**repeat**
3	Take an action according to π and observe r, s'
4	$\theta \leftarrow \theta + \alpha\phi(s)\left[r + \left(\gamma\phi(s') - \phi(s)\right)^T \theta\right]$
5	end repeat

Algorithm 1: TD(0)

in simulated soccer, there were over ten thousand features but the number of non-zero features for any state was only 416.

Because the policy's true value function is probably not in our space of linear functions, we want to find a set of parameters that approximates the true function. One possible approach is to use the observed TD error on sample trajectories of experience to guide the approximation. Both TD(0) and LSTD take this approach.

Temporal Difference Learning

The standard one-step TD method for value function approximation is TD(0).[1] The basic idea of TD(0) is to adjust a state's predicted value to reduce the observed TD error. Given some new experience tuple $(s_t, a_t, r_{t+1}, s_{t+1})$, the update with linear function approximation is,

$$
\begin{aligned}
\theta_{t+1} &= \theta_t + \alpha_t \mathbf{u}_t(\theta_t), \text{ where} \\
\mathbf{u}_t(\theta) &= \phi(s_t)\delta_t(V_\theta). \quad (3)
\end{aligned}
$$

V_θ is the estimated value with respect to θ and α_t is the learning rate. The vector $\mathbf{u}_t(\theta)$ is like a gradient estimate that specifies how to change the predicted value of s_t to reduce the observed TD error. We will call $\mathbf{u}_t(\theta)$ the TD update at time t. After updating the parameter vector, the experience tuple is forgotten. Pseudocode for TD(0) is shown above as Algorithm 1.

The computational costs of TD(0) for each time step is due mainly to the vector addition, which is linear in the length of the vector, *i.e.*, $O(n)$. If only k features are non-zero for any state, then a sparse vector representation can further reduce the computation to $O(k)$. So, the algorithm is (sub)-linear in the number of features, which allows it to be applied even with very large feature representations.

Least-Squares TD

The Least-Squares TD algorithm (LSTD) (Bradtke & Barto 1996) can be seen as immediately solving for the value function parameters for which the sum TD update over all the observed data is zero. Let $\boldsymbol{\mu}_t(\theta)$ be the sum of the TD updates over the data through time t,

$$\boldsymbol{\mu}_t(\theta) = \sum_{i=1}^{t} \mathbf{u}_i(\theta). \quad (4)$$

---

[1] Although in this paper we treat only one-step methods, our ideas can probably be extended to multi-step methods such as TD(λ) and LSTD(λ).

```
0    s ← s_0, A ← 0, b ← 0
1    Initialize θ arbitrarily
2    repeat
3       Take action according to π and observe r, s′
5       b ← b + φ(s)r
6       d ← (φ(s) − γφ(s′))
7       A ← A + φ(s)d^T
8       if (first update)
9          Ã ← A^{-1}
10      else
11         Ã ← Ã( I − ( φ(s)d^T / (1 + d^T Ãφ(s)) ) Ã )
12      end if
13      θ ← Ãb
14   end repeat
```

<div align="center">Algorithm 2: LSTD</div>

Let $\phi_t = \phi(s_t)$. Applying Equations 3 and 2, and the definition of our linear value functions, we get,

$$
\begin{aligned}
\boldsymbol{\mu}_t(\boldsymbol{\theta}) &= \sum_{i=1}^{t} \phi_t \delta_t(V_{\boldsymbol{\theta}}) \\
&= \sum_{i=1}^{t} \phi_t \left(r_{t+1} + \gamma \phi_{t+1}^T \boldsymbol{\theta} - \phi_t^T \boldsymbol{\theta} \right) \\
&= \sum_{i=1}^{t} \left(\phi_t r_{t+1} - \phi_t (\phi_t - \gamma \phi_{t+1})^T \boldsymbol{\theta} \right) \\
&= \left(\underbrace{\sum_{i=1}^{t} \phi_t r_{t+1}}_{\mathbf{b}_t} - \underbrace{\sum_{i=1}^{t} \phi_t (\phi_t - \gamma \phi_{t+1})^T}_{\mathbf{A}_t} \boldsymbol{\theta} \right) \\
&= (\mathbf{b}_t - \mathbf{A}_t \boldsymbol{\theta}). \qquad (5)
\end{aligned}
$$

Since we want to choose parameters such that the sum TD update is zero, we set Equation 5 to zero and solve for the new parameter vector,

$$ \boldsymbol{\theta}_{t+1} = \mathbf{A}_t^{-1} \mathbf{b}_t. $$

The online version of LSTD incorporates each observed reward and state transition into the **b** vector and the **A** matrix and then solves for a new $\boldsymbol{\theta}$. Notice that, once **b** and **A** are updated, the experience tuple can be forgotten without losing any information. Because **A** changes by only a small amount on each time step, $\mathbf{A}^{-1}$ can also be maintained incrementally. This version of LSTD is shown above as Algorithm 2.

LSTD after each time step computes the value function parameters that have zero sum TD update. It essentially fully exploits all of the observed data to compute its approximation. However, this data efficiency is at the cost of computational efficiency. In the non-incremental form, the matrix inversion alone is $O(n^3)$. Using the incremental form, maintaining the matrix inversion still requires $O(n^2)$ computation per time step. If the feature vector is sparse, some of the operations such as updating **A** become $O(k^2)$, but others such as multiplying $\mathbf{A}^{-1}\mathbf{b}$ are still $O(n^2)$ because neither the matrix nor the vector are necessarily sparse. The result is that LSTD can be computationally impractical for problems with a large number of features even if they are sparse.

Practitioners are currently faced with a serious tradeoff: they must choose between data efficiency or computational efficiency. In the next section, we introduce a new algorithm that seeks to provide a compromise between these extremes. In particular, our algorithm exploits all of the data like LSTD, while requiring only linear computation per time step when a small number of features are non-zero.

New Algorithm

In this section we present the incremental least-squares temporal difference learning algorithm (iLSTD). The algorithm computes and uses the sum TD update over all observed trajectories, thus making more efficient use of the data than TD. However, iLSTD does not immediately solve for the parameters that give a zero sum TD update, which is too computationally expensive. Instead, the sum TD update is used in a gradient fashion to move the parameters in the direction to reduce it to zero.

Incremental Computation

The key step in iLSTD is incrementally computing $\boldsymbol{\mu}_t(\boldsymbol{\theta})$ as transitions are observed and $\boldsymbol{\theta}$ changes. We begin by showing that **b** and **A** can be computed in an incremental fashion given a newly observed reward and transition:

$$
\begin{aligned}
\mathbf{b}_t &= \mathbf{b}_{t-1} + \underbrace{r_t \phi_t}_{\Delta \mathbf{b}_t} \\
\mathbf{A}_t &= \mathbf{A}_{t-1} + \underbrace{\phi_t (\phi_t - \gamma \phi_{t+1})^T}_{\Delta \mathbf{A}_t}.
\end{aligned}
$$

Given a new **b** and **A**, we can incrementally compute $\boldsymbol{\mu}_t(\boldsymbol{\theta}_t)$:

$$ \boldsymbol{\mu}_t(\boldsymbol{\theta}_t) = \boldsymbol{\mu}_{t-1}(\boldsymbol{\theta}_t) + \Delta \mathbf{b}_t - (\Delta \mathbf{A}_t)\boldsymbol{\theta}_t. $$

Finally, we can incrementally compute $\boldsymbol{\mu}_t(\boldsymbol{\theta}_{t+1})$, given an update $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \Delta \boldsymbol{\theta}_t$, as:

$$ \boldsymbol{\mu}_t(\boldsymbol{\theta}_{t+1}) = \boldsymbol{\mu}_t(\boldsymbol{\theta}_t) - \mathbf{A}_t(\Delta \boldsymbol{\theta}_t). \qquad (6) $$

We will examine the time complexity of these computations after presenting the complete algorithm.

Updating the Parameters

We've observed that solving for $\boldsymbol{\theta}_{t+1}$ such that $\boldsymbol{\mu}_t(\boldsymbol{\theta}_{t+1}) = 0$ requires quadratic time in the number of features. Instead we might consider taking a step in the direction of $\boldsymbol{\mu}$. This can be thought of as applying the total change to $\boldsymbol{\theta}$ if TD were applied to all transitions from our previously observed trajectories in a batch fashion. As such, it both makes use of all the past data and will have a lower variance than TD's traditional single sample update. Unfortunately, it too is computationally expensive, as Equation 6 takes $O(n^2)$ to compute if $\Delta \boldsymbol{\theta}_t$ has few zero components.

```
0    s ← s₀, **A** ← **0**, **μ** ← **0**, t ← 0
1    Initialize **θ** arbitrarily
2    **repeat**
3       Take action according to π and observe r, s'
4       t ← t + 1
5       Δ**b** ← φ(s)r
6       Δ**A** ← φ(s)(φ(s) − γφ(s'))ᵀ
7       **A** ← **A** + Δ**A**
8       **μ** ← **μ** + Δ**b** − (Δ**A**)**θ**
9       **for** i from 1 to m **do**
10         j ← argmax(|μⱼ|)
11         θⱼ ← θⱼ + αμⱼ
12         **μ** ← **μ** − αμⱼ**A**eᵢ
13      **end for**
14   **end repeat**
```

Algorithm 3: iLSTD

The lack of many zero components in $\Delta\boldsymbol{\theta}_t$ suggests the compromise that will be used by iLSTD. Instead of updating all of $\boldsymbol{\theta}$, iLSTD only considers updating a small number of components of $\boldsymbol{\theta}$. For example, consider updating only the ith components:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha_t \mu_t(i)\boldsymbol{e}_i$$
$$\boldsymbol{\mu}_t(\boldsymbol{\theta}_{t+1}) = \boldsymbol{\mu}_t(\boldsymbol{\theta}_t) - \alpha_t \mu_t(i)\mathbf{A}_t\boldsymbol{e}_i,$$

where $\mu_t(i)$ is the ith component of $\boldsymbol{\mu}_t$ and $\boldsymbol{e}_i$ is the column vector with a single one in the ith row (thus $\mathbf{A}_t\boldsymbol{e}_i$ selects the ith column of the matrix $\mathbf{A}_t$). Multiple components can be updated by repeatedly selecting a component and performing the one component update above. Our algorithm takes a parameter $m \ll n$ that specifies the number of updates that are performed per time step.

What remains is to select which components to update. Because we want to select a component that will most reduce the sum TD update, we can simply choose the component with the *largest* sum TD update using the values maintained in $\boldsymbol{\mu}_t(\boldsymbol{\theta}_t)$. This approach has a resemblance to prioritized sweeping (Moore & Atkeson 1993), but rather than updating the *state* with the largest TD update, we choose to update the *parameter component* with the largest TD update. Like prioritized sweeping, iLSTD can tradeoff data efficiency and computational efficiency by increasing m, the number of components updated per time step.

Algorithm

Algorithm 3 gives the complete iLSTD algorithm. After setting the initial values, the agent begins interacting with the environment. $\mathbf{A}$ and $\boldsymbol{\mu}$ are computed incrementally in Lines 5–8.[2] It is followed by updating selected parameter components in Lines 9–13. For each of the m updates performed during the single time step, the component with the highest absolute value of the sum TD update vector ($\boldsymbol{\mu}_t$) is chosen. After the update, $\boldsymbol{\mu}$ is recomputed and so may affect the next component chosen.

---

[2]$\mathbf{b}$ is not computed because it is implicitly included in $\boldsymbol{\mu}$.
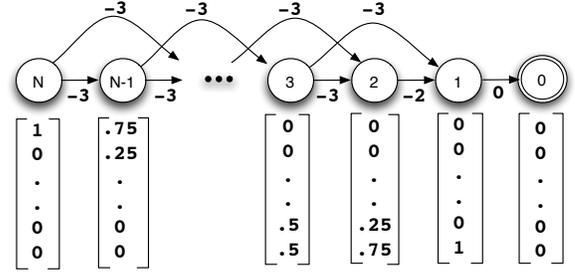
Figure 1: The extended Boyan chain example with a variable number of states.

Time Complexity

We now examine iLSTD's time complexity.

Theorem 1 *If there are n features and for any given state s, $\phi(s)$ has at most k non-zero elements, then the iLSTD algorithm is $O(mn + k^2)$ per time step.*

Proof Lines 6–8 are the most computationally expensive parts of iLSTD outside the inner loop. Because each feature vector does not have more than k non-zero elements, $\phi(s)r$ has only k non-zero elements and the matrix $\phi(s)(\phi(s) - \gamma\phi(s'))^T$ has at most $2k^2$ non-zero elements. Therefore lines 6-8 are computable in $O(k^2)$ with sparse matrices and vectors. Inside the parameter update loop (Line 9), the expensive lines are 10 and 12. The argmax can be computed in $O(n)$ and because $\mathbf{A}\boldsymbol{e}_i$ is just the $i$the column of $\mathbf{A}$, the parameter update is also $O(n)$. This will lead to $O(mn + k^2)$ as the final bound for the algorithm per time step. $\square$

Empirical Results

The experimental results in this paper follow closely with Boyan's experiments with LSTD(λ) (Boyan 1999). The problem we examine is the Boyan chain problem: Figure 1 depicts the problem in the general form. In order to demonstrate the computational complexity of iLSTD, results were conducted with three different problem sizes: 14 (original problem), 102 and 402 states. We will call these the small, medium, and large problems, respectively. It is an episodic problem, starting at state N and being terminated in state zero. For all states, $s > 2$, there exists equal probability of ending up in states $(s-1)$ or $(s-2)$, and reward of all transitions are -3, except from state 2 to 1 (when reward is -2) and transitions to state 0 (when reward is 0).

The α step size used in these experiments takes the same form as that used in Boyan's original experiments.

$$\alpha_t = \alpha_0 \frac{N_0 + 1}{N_0 + \text{Episode\#}}$$

The selection of N_0 and α_0 for TD and iLSTD was based on experimentally finding the best parameters in the set $\alpha_0 \in \{0.01, 0.1, 1\}$ and $N_0 \in \{100, 1000, 10^6\}$. We only report the results for the best set of parameters for each algorithm. The m parameter for iLSTD was set to one, so on each time step only one component was updated.
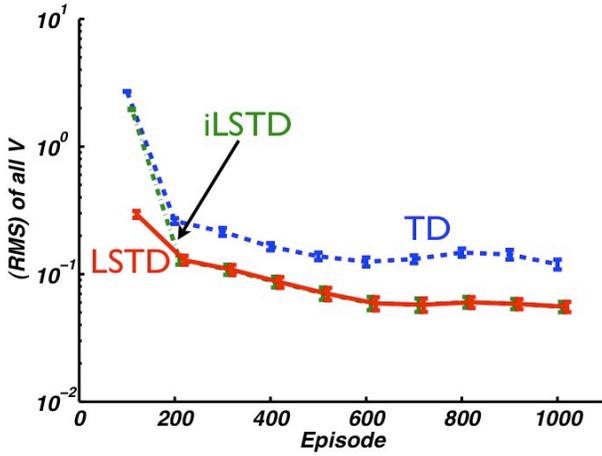
Figure 2: Results on the *small* problem (States = 14, Features = 4).

Figure 3: Results on the *medium* problem (States = 102, Features = 26).

The performance of all three algorithms on the small, medium, and large problems, averaged over 30 runs, are shown in Figures 2, 3, and 4, respectively. For each run the algorithms were given the same trajectories of data. The vertical axis shows the average RMS error value of all states in a log scale, while the horizontal axis shows the number of episodes. The vertical bars show the confidence interval for the performance and are shifted a small number of episodes in order to make them more visible. As the complexity of the problem increases the gap between the two least-squares algorithms and TD gets considerably wider. In addition, iLSTD's performance follows very closely to LSTD. The relative difference is most dramatic in the large problem, suggesting that the data efficiency of the least-squares approaches is more pronounced in larger problems.

Figure 4: Results on the *large* problem (States = 402, Features = 101).

Running Time

All three methods were implemented using sparse vector algebra to take advantage of the sparsity of $\phi(s)$ and $\mathbf{A}$. They were also compared to non-sparse versions to insure that the overhead of sparse vectors did not actually result in slower implementations. We performed timing experiments on a single 3.06 GHz Intel Xenon processor with increasing number of states. The results were consistent with our timing analyses and Theorem 1. Figure 5 shows the total running time of 1000 episodes on a log scale.[3]

Discussion

iLSTD occupies the middle ground between the computational efficiency of TD and the data efficiency of LSTD. When only a small number of features are non-zero, iLSTD's asymptotic computational complexity is only linear in the number of features, while LSTD is quadratic. Our experimental results confirm this on simple problems. In these experiments, we also observed iLSTD's data efficiency was on par with LSTD, and a vast improvement over TD.

In situations where data is overwhelmingly more expensive than computation, LSTD makes the most sense. In situations where computation is overwhelmingly more expensive than data, TD makes the most sense.[4] But most problems reside in the middle ground where neither data nor computation are prohibitively costly, for which iLSTD may be the best choice. Even when data is free, iLSTD can still outperform other methods. In all of the problem sizes, iLSTD took less wall-clock time to reach a strong level of performance. For example, although TD was twice as fast

---

[3]The results for more then 101 features are based on a single run of identical data.

[4]Notice that large problems with many features effectively raise the cost of computation, as even simple vector algebra with large non-sparse vectors can be time consuming. Thus for problems with millions or billions of features, TD with its $O(k)$ per time step complexity likely remains the only viable option.
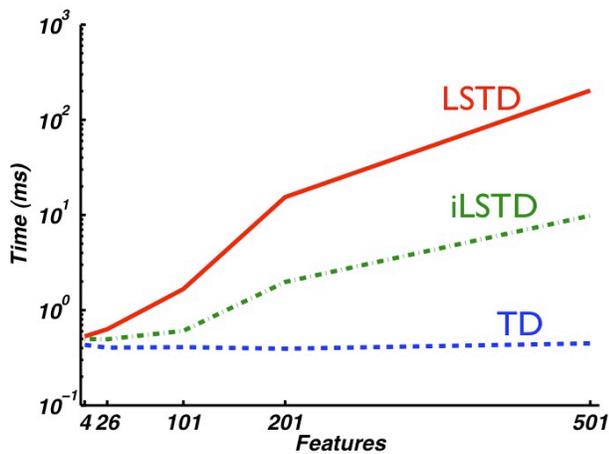
Figure 5: Running time of each method per time-step with respect to the problem size.

computationally as iLSTD, its performance with twice as much data was inferior.

Conclusions

Our experiments show that as the number of features grow, computationally expensive methods like LSTD can become impractical. However, computationally cheap methods like TD do not make efficient use of the data. We introduced iLSTD as an alternative when neither computation nor data are prohibitively expensive. If the number of non-zero features at any time step is small, iLSTD's computation grows only linearly with the number of features. In addition, our experiments show that iLSTD's performance is comparable to that of LSTD.

There are a number of interesting directions for future work. First, the convergence of iLSTD is an open question. Whether it, in general, converges to the same parameters as TD and LSTD is also unknown. Second, eligibility traces have proven to be effective at improving the performance of TD methods, giving birth to the TD(λ) (Sutton 1988) and LSTD(λ) (Boyan 1999) algorithms. It would be interesting to see if they can be used with iLSTD without increasing computational complexity. Finally, we would like to apply the iLSTD algorithm to a more challenging problem with a large number of features for which LSTD is currently infeasible such as keepaway in simulated soccer (as in Stone, Sutton, & Kuhlmann 2005).

Acknowledgments

References

Albus, J. S. 1971. A theory of cerebellar function. *Mathematical Biosciences* 10:25–61.

Boyan, J. A. 1999. Least-squares temporal difference learning. In *Proceedings of the Sixteenth International Conference on Machine Learning*, 49–56. Morgan Kaufmann, San Francisco, CA.

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

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

Hinton, G. E.; McClelland, J. L.; and Rumelhart, D. E. 1986. Distributed representations. In Rumelhart, D. E.; and McClelland, J. L., eds., *Parallel Distributed Processing: Explorations in the Microstructure of Cognition. Volume 1: Foundations*.

Lin, L. J. 1993. *Reinforcement Learning for Robots Using Neural Networks*. Ph.D. Dissertation, Carnegie Mellon University.

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

Poggio, T., and Girosi, F. 1990. Networks for approximation and learning. *Proceedings of the IEEE (special issue: Neural Networks I: Theory and Modeling)* 78(9):1481–1497.

Stone, P.; Sutton, R. S.; and Kuhlmann, G. 2005. Reinforcement learning for robocup soccer keepaway. *International Society for Adaptive Behavior* 13(3):165–188.

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

Sutton, R. S. 1988. Learning to predict by the methods of temporal differences. *Machine Learning* 3:9–44.

Sutton, R. S. 1996. Generalization in reinforcement learning: Successful examples using sparse coarse coding. In Touretzky, D. S.; Mozer, M. C.; and Hasselmo, M. E., eds., *Advances in Neural Information Processing Systems 8*, 1038–1044. The MIT Press.

Tedrake, R.; Zhang, T. W.; and Seung, H. S. 2004. Stochastic policy gradient reinforcement learning on a simple 3D biped. In *Proceedings of the IEEE International Conference on Intelligent Robots and Systems*, 2849–2854.

Tsitsiklis, J. N., and Van Roy, B. 1997. An analysis of temporal-difference learning with function approximation. *IEEE Transactions on Automatic Control* 42(5):674–690.

Xu, X.; He, H.; and Hu, D. 2002. Efficient reinforcement learning using recursive least-squares methods. *Journal of Artificial Intelligence Research* 16:259–292.