Research Article
Kernel Recursive Least-Squares Temporal Difference Algorithms with Sparsification and Regularization

Chunyuan Zhang,1,2 Qingxin Zhu,1 and Xinzheng Niu1

1 School of Computer Science and Engineering, University of Electronic Science and Technology of China, Chengdu 611731, China
2 College of Information Science and Technology, Hainan University, Haikou 570228, China

Correspondence should be addressed to Chunyuan Zhang; zcy7566@126.com

Received 18 January 2016; Revised 23 April 2016; Accepted 28 April 2016

Academic Editor: Manuel Graña

Copyright © 2016 Chunyuan Zhang et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

By combining with sparse kernel methods, least-squares temporal difference (LSTD) algorithms can construct the feature dictionary automatically and obtain a better generalization ability. However, the previous kernel-based LSTD algorithms do not consider regularization and their sparsification processes are batch or offline, which hinder their widespread applications in online learning problems. In this paper, we combine the following five techniques and propose two novel kernel recursive LSTD algorithms: (i) online sparsification, which can cope with unknown state regions and be used for online learning, (ii) $L_2$ and $L_1$ regularization, which can avoid overfitting and eliminate the influence of noise, (iii) recursive least squares, which can eliminate matrix-inversion operations and reduce computational complexity, (iv) a sliding-window approach, which can avoid caching all history samples and reduce the computational cost, and (v) the fixed-point subiteration and online pruning, which can make $L_1$ regularization easy to implement. Finally, simulation results on two 50-state chain problems demonstrate the effectiveness of our algorithms.

1. Introduction

The least-squares temporal difference (LSTD) learning may be the most popular approach for policy evaluation in reinforcement learning (RL) [1, 2]. Compared with the standard temporal difference (TD) learning, LSTD uses samples more efficiently and eliminates all step-size parameters. However, LSTD also has some drawbacks. First, LSTD requires a matrix-inversion operation at each time step. To reduce computational complexity, Bradtke and Barto proposed a recursive LSTD (RLSTD) algorithm [1], and Xu et al. proposed a RLSTD($\lambda$) algorithm [3]. But these two algorithms still require many features especially for highly nonlinear RL problems, since the RLS approximator assumes a linear model [4]. Second, when the number of features is larger than the number of training samples, LSTD is prone to overfitting. To overcome this problem, Kolter and Ng proposed an $L_1$-regularized LSTD algorithm called LARS-TD for feature selection [5], but it is only applicable for batch learning and its implementation is complicated. On this basis, Chen et al. proposed an $L_2$-regularized RLSTD algorithm [6]. In contrast with LARS-TD, it has an analytical solution, but it cannot obtain a sparse solution. Third, LSTD requires users to design the feature vector manually, and poor design choices can result in estimates that diverge from the optimal value function [7].

In the last two decades, kernel methods have been intensively and extensively studied in supervised and unsupervised learning [8]. The basic idea behind kernel methods can be summarized as follows: By using a nonlinear transform, the origin input data can be mapped into a high-dimensional feature space, and an inner product in this space can be interpreted as a Mercer kernel function. Thus, as long as a linear algorithm can be formulated in terms of inner products, there is no need to perform computations in the high-dimensional feature space [9]. Recently, there have also been many research works on kernelizing least-squares algorithms [9–13]. Here, we only review some works related to our proposed algorithms. One typical work is the sparse kernel recursive least-squares (SKRLS) algorithm with the approximate linear dependency (ALD) criterion [11]. Compared with traditional RLS algorithms, it not only has
a good nonlinear approximation ability but also can construct the feature dictionary automatically. Similarly, Chen et al. proposed an $L_2$-regularized SKRLS algorithm with the online vector quantization [12]. Besides having the good properties of SKRLS-ALD, it can avoid overfitting. In addition, Chen et al. proposed an $L_1$-regularized SKRLS algorithm with the fixed-point subiteration [13], which can yield a much sparser dictionary.

Intuitively, we can also bring the benefits of kernel machine learning to LSTD algorithms. In fact, kernel-based RL algorithms have become more and more popular in recent years [14–22], and several works have been done for kernelizing LSTD algorithms. In an earlier paper, Xu proposed a sparse kernel-based LSTD(λ) (SKLSTD(λ)) algorithm with the ALD criterion [19]. Although this algorithm can avoid selecting features manually, it is only applicable for batch learning and its derivation is complicated. After that, Xu et al. proposed an incremental version of the SKLSTD(λ) algorithm for policy iteration [20], but this algorithm still requires a matrix-inversion operation at each time step. Moreover, the feature dictionary is required to be constructed offline, which makes this algorithm only approximate the value function correctly in the area of the state space that is covered by the training samples. Recently, Jakab and Csató proposed a sparse kernel RLSTD (SKRLSTD) algorithm by using a proximity graph sparsification method [21]. Unfortunately, its sparsification process is also offline. In addition, all of these algorithms do not consider regularization, whereas many real problems exhibit noise and the high expressiveness of the kernel matrix can result in overfitting [22].

In this paper, we propose two online SKRLSTD algorithms with $L_2$ and $L_1$ regularization, called OSKRLSTD-$L_2$ and OSKRLSTD-$L_1$, respectively. Compared with the derivation of SKLSTD(λ), our derivation uses Bellman operator along with projection operator and thus is more simple. To cope with unknown state-space regions and avoid overfitting, our algorithms use online sparsification and regularization techniques. Besides, to reduce computational complexity and avoid caching all history samples, our algorithms also use the recursive least-squares and the sliding-window technique. Moreover, different from LARS-TD, OSKRLSTD-$L_1$ uses the subiteration and online pruning to find the fixed point. These techniques make our algorithms more suitable for online RL problems with a large or continuous state space. The rest of this paper is organized as follows. In Section 2, we present preliminaries and review the LSTD algorithm. Section 3 contains the main contribution of this paper: we derive OSKRLSTD-$L_2$ and OSKRLSTD-$L_1$ algorithms in detail. In Section 4, we demonstrate the effectiveness of our algorithms for two 50-state chain problems. Finally, we conclude the paper in Section 5.

2. Background

In this section, we introduce the basic definitions and notions, which will be used throughout the paper without any further mention. We also review the LSTD algorithm, which is needed to establish our algorithms described in Section 3.

2.1. Preliminaries. In RL and dynamic programming (DP), an underlying sequential decision-making problem is often modeled as a Markov decision process (MDP). An MDP can be defined as a tuple $\mathcal{M}=⟨\mathcal{S}, \mathcal{A}, P, r, γ, d⟩$ [5], where $\mathcal{S}$ is a set of states, $\mathcal{A}$ is a set of actions, $P : \mathcal{S} × \mathcal{A} × \mathcal{S} → [0, 1]$ is a state transition probability function where $P(s, a, s')$ denotes the probability of transitioning to state $s'$ when taking action $a$ in state $s$, $r \in \mathbb{R}$ is a reward function, $γ \in [0, 1]$ is the discount factor, and $d$ is an initial state distribution. For simplicity of presentation, we assume that $\mathcal{S}$ and $\mathcal{A}$ are finite. Given an MDP $\mathcal{M}$ and a policy $\pi : \mathcal{S} → \mathcal{A}$, the sequence $s_1, r_1, s_2, r_2, \ldots$ is a Markov reward process $\mathcal{R}=⟨\mathcal{S}, P, R, γ, d⟩$, where $P^T(s, s') = \sum_{a\in\mathcal{A}} \pi(a | s)P(s, a, s')$ and $R^\pi(s) = \sum_{a\in\mathcal{A}} \pi(a | s)\sum_{s'\in\mathcal{S}} P(s, a, s')r(s, a, s')$.

RL and DP often use the state-value function $V^\pi(s)$ to evaluate how good the policy $\pi$ is for the agent to be in state $s$. For an MDP, $V^\pi(s)$ can be defined as $V^\pi(s) = E_{\mathcal{R}}\left[\sum_{t=0}^{\infty} γ^tr_t | s_0 = s\right]$, which must obey the Bellman equation [23],

$$V^\pi(s) = R^\pi(s) + γ \sum_{s' \in \mathcal{S}} P^\pi(s, s') V^\pi(s'),$$

(1)

or be expressed in vector form,

$$V^\pi = R^\pi + γ P^\pi V^\pi.$$

(2)

If $P^\pi$ and $R^\pi$ are known, $V^\pi$ can be solved analytically; that is,

$$V^\pi = (I - γ P^\pi)^{-1} R^\pi,$$

(3)

where $I$ is the $|\mathcal{S}| \times |\mathcal{S}|$ identity matrix.

However, different from the case in DP, $P^\pi$ and $R^\pi$ are unknown in RL. The agent has to estimate $V^\pi$ by exploring the environment. Furthermore, many real problems have a large or continuous state space, which makes $V^\pi(s)$ hard to be expressed explicitly. To overcome this problem, we often resort to linear function approximation; that is,

$$\hat{V}^\pi(s) = \phi^T(s) w,$$

(4)

or

$$\hat{V}^\pi = \Phi w,$$

where $w \in \mathbb{R}^m$ is a parameter vector, $\phi(s) \in \mathbb{R}^m$ is the feature vector of state $s$, and $\Phi = [\phi(s_1), \ldots, \phi(s_n)]^T$ is a $|\mathcal{S}| \times m$ feature matrix. Unfortunately, when approximating $V^\pi$ in this manner, there is usually no way to satisfy the Bellman equation exactly, because $R^\pi + γ P^\pi \Phi w$ may lie outside the span of $\Phi$ [5].

2.2. LSTD Algorithm. The LSTD algorithm presents an efficient way to find $w$ such that $\hat{V}^\pi$ “approximately” satisfies the Bellman equation [5]. By solving the least-squares problem $\min_{w \in \mathbb{R}^m} \| \Phi w - (R^\pi + γ P^\pi \Phi w) \|^2_D$, we can find a closest approximation $\Phi \hat{w}$ in the span of $\Phi$ to replace $R^\pi + γ P^\pi \Phi w$. Then, from (2) and (4), we can use $w = \hat{w}$ for approximating $V^\pi$. That means we can find $w$ by solving the fixed-point equation:

$$w = f(w) = \arg \min_{w \in \mathbb{R}^m} \| \Phi w - (R^\pi + γ P^\pi \Phi w) \|^2_D.$$

(5)
where $D$ is a nonnegative diagonal matrix indicating a distribution over states. Nevertheless, since $P^T$ and $R^T$ are unknown and since $\Phi$ is too large to form anyway in a large or continuous state space, we cannot solve (5) exactly. Instead, given a trajectory $T^* = \{(s_i, s'_i, r_i) | i = 1, \ldots, t\}$ following policy $\pi$, LSTD uses $\hat{\Phi} = [\phi(s_1), \ldots, \phi(s_t)]^T$, $\hat{\Phi}' = [\phi(s'_1), \ldots, \phi(s'_t)]^T$, and $\hat{R} = [r_1, \ldots, r_t]^T$ to replace $\Phi$, $P^\pi \Phi$, and $R^\pi$, respectively. Then, (5) can be approximately rewritten as

$$w_t = \tilde{f}(w_t) = \arg \min_{u \in \mathbb{R}^n} \| \hat{\Phi} u - (\hat{R} + y\hat{\Phi}' w_t) \|_2^2$$

(6)

Let $\partial \| \hat{\Phi} u - (\hat{R} + y\hat{\Phi}' w_t) \|_2^2 / \partial u = 0$; we have

$$\tilde{f}(w_t) = u^* = (\hat{\Phi}_1^T \hat{\Phi}_1)^{-1} \hat{\Phi}_1^T (\hat{R} + y\hat{\Phi}' w_t).$$

(7)

Thus, the fixed point $w_t = \tilde{f}(w_t)$ can be found by

$$w_t = (\hat{\Phi}_1^T (\hat{\Phi}_1 - y\hat{\Phi}'_1))^{-1} \hat{\Phi}_1^T \hat{R}.$$  

(8)

### 3. Regularized OSKRLSTD Algorithms

To overcome the weaknesses of the previous kernel-based LSTD algorithms, we propose two regularized OSKRLSTD algorithms in this section.

#### 3.1. OSKRLSTD-L2 Algorithm

Now, we use $L_2$ regularization and online sparsification to derive the first OSKRLSTD algorithm, which is called OSKRLSTD-L2.

First, we use the kernel trick to kernelize (6). Suppose the feature dictionary $\mathcal{D}_j = \{d_j | d_j \in \mathcal{S}, j = 1, \ldots, n\}$, and let $\Phi_j = [\phi(d_1), \ldots, \phi(d_{n})]^T$ denote the corresponding feature matrix. By the Representer Theorem [24], $w_t$ and $u$ can be expressed as follows:

$$w_t = \Phi_j^T \alpha_t = \sum_{j=1}^{n_t} \alpha_j \phi(d_j),$$

$$u = \Phi_j^T \beta = \sum_{j=1}^{n_t} \beta_j \phi(d_j),$$

(9)

where $\alpha_t = [\alpha_{t,1}, \ldots, \alpha_{t,n_t}]^T$ and $\beta = [\beta_1, \ldots, \beta_{n_t}]^T$ are the coefficient vector of $w_t$ and $u$, respectively. Then, from (6), we have

$$\alpha_t = \tilde{f}(\alpha_t) = \arg \min_{\beta \in \mathbb{R}^{n_t}} \| \Phi_j \Phi_j^T \beta - (\hat{R} + y\hat{\Phi}'_j \Phi_j^T \alpha_t) \|_2^2.$$  

(10)

By the Mercer Theorem [24], the inner product of two feature vectors can be calculated by $k(s, s') = \phi(s)^T \phi(s')$. Thus, we can define $K = \Phi_j \Phi_j^T$, $\hat{K} = \hat{\Phi}_1 \hat{\Phi}_1^T$, and $\hat{K}' = \hat{\Phi}'_1 \hat{\Phi}'_1^T$. On this basis, (10) can be rewritten as

$$\alpha_t = \tilde{f}(\alpha_t) = \arg \min_{\beta \in \mathbb{R}^{n_t}} \| \hat{K} \beta - (\hat{R} + y \hat{K}' \alpha_t) \|_2^2.$$  

(11)

Second, we try to derive the $L_2$-regularized solution of (II). Add an $L_2$-norm penalty into (II); that is,

$$\begin{align*}
\alpha_t &= \tilde{f}(\alpha_t) \\
&= \arg \min_{\beta \in \mathbb{R}^{n_t}} \left\| \hat{K} \beta - (\hat{R} + y \hat{K}' \alpha_t) \right\|_2^2 + \eta \| \beta \|_2^2,
\end{align*}$$

(12)

where $\eta \in (0, \infty)$ is a regularization parameter. Let $\partial \| \hat{K} \beta - (\hat{R} + y \hat{K}' \alpha_t) \|_2^2 / \partial \beta = 0$; we have

$$\hat{K} (\hat{K} \beta - (\hat{R} + y \hat{K}' \alpha_t)) + \eta \beta^* = \hat{K} \hat{R},$$

(13)

Since $w_t = u^*$, we easily have $\alpha_t = \beta^*$ from (9). Then, the above equation can be rewritten as

$$\hat{K} \left( \hat{K} \beta^* - (\hat{R} + y \hat{K}' \alpha_t) \right) + \eta I_n \alpha_t = \hat{K} \hat{R},$$

(14)

where $I_n$ is the $n_t \times n_t$ identity matrix. Thus, $\alpha_t$ can be analytically solved as

$$\alpha_t = \left( \hat{K} \left( \hat{K} - y \hat{K}' \right) + \eta I_n \right)^{-1} \hat{K} \hat{R},$$

(15)

where $\hat{A}_t \in \mathbb{R}^{n_t \times n_t}$ and $\hat{b}_t \in \mathbb{R}^{n_t}$ denote

$$\hat{A}_t = \hat{K} \left( \hat{K} - y \hat{K}' \right) + \eta I_n,$$

$$\hat{b}_t = \hat{K} \hat{R},$$

(16)

where $k(\cdot, d_1), \ldots, k(\cdot, d_{n_t})^T$ and $\Delta k(s, s') = k(s, s') - yk(s, s')$.

Third, we derive the recursive formulas of $A_t^{-1}$ and $\alpha_t$. Under online sparsification, there are two cases: (1) $\mathcal{D}_j = \mathcal{D}_j$, $n_t = n_{j-1}$, $\hat{k}(\cdot) = k(\cdot, d_{n_{j-1}})$, $\Delta k(s, s') = \Delta k(\cdot, d_{n_{j-1}})(s, s')$, and $I_t = I_{t-1}$; (2) $\mathcal{D}_j = \mathcal{D}_j - \{s_{n_{j-1}}\} \cup \{s_{n_{t}}\}$, $n_t = n_{j-1} + 1$, $k(\cdot) = [k_{n-1}(\cdot), k(\cdot, s_t)]^T$, $\Delta k(s, s') = \Delta k(\cdot, d_{n_{j-1}})(s, s')$, $\Delta k(\cdot, d_{n_{j-1}})(s, s')$, and $I_t$ is expanded as

$$I_t = \begin{bmatrix} I_{t-1} & 0_{t-1} \\ 0_{t-1}^T & 1 \end{bmatrix},$$

(17)

where $0_{t-1}$ is the $n_{t-1}$ dimensional zero vector.

For the first case, (16) can be rewritten as follows:

$$\hat{A}_t = A_{t-1} + \hat{k}_{n_t-1}(s_t) \Delta k_{n_t-1} \left( s_t, s'_t \right),$$

$$\hat{b}_t = b_{t-1} + k_{n_t-1}(s_t) r_t.$$

(18)

(19)

Applying the matrix-inversion lemma [25] for $A_t^{-1}$, we get

$$A_t^{-1} = A_{t-1}^{-1} - \frac{A_{t-1}^{-1} \hat{k}_{n_t-1}(s_t) \Delta k_{n_t-1} \left( s_t, s'_t \right) A_{t-1}^{-1}}{1 + \Delta k_{n_t-1} \left( s_t, s'_t \right) A_{t-1}^{-1} \hat{k}_{n_t-1}(s_t)},$$

(20)
Thus, plugging (19) and (20) into (15), we obtain
\[
\alpha_t = \alpha_{t-1} + \frac{(r_t - \alpha_{t-1}^T \Delta k_{t-1}(s_t, s_t')) A_{t-1}^T k_{t-1}(s_t)}{1 + \Delta k_{t-1}^T(s_t, s_t') A_{t-1}^{-1} k_{t-1}(s_t)}.
\] (21)

For the second case, (16) can be rewritten as follows:
\[
A_t = \begin{bmatrix} \mathbf{A}_t & h_t \\ \mathbf{g}_t & \mathbf{p}_t \end{bmatrix},
\]
\[
b_t = \begin{bmatrix} b_t \\ q_t \end{bmatrix},
\] (22)

where \(\mathbf{A}_t\) and \(\mathbf{b}_t\) are the same as the updated \(\mathbf{A}_t\) and \(\mathbf{b}_t\) when the feature dictionary keeps unchanged, \(h_t = \sum_{i=1}^{t-1} \Delta k_i(s_i, s_i') k_{t-1}(s_t)\), \(g_t = \sum_{i=1}^{t-1} k(s_i, s_i) \Delta k_{t-1}(s_t, s_t')\), \(p_t = \sum_{i=1}^{t-1} k(s_i, s_i) \Delta k_i(s_t, s_t') + \eta_t\) and \(q_t = \sum_{i=1}^{t-1} k(s_i, s_i) r_t\). However, computing \(h_t, g_t, p_t\), and \(q_t\) requires caching all history samples, and the computational cost will become more and more expensive as \(t\) increases. Inspired by the work of Van Vaerenbergh et al. [26], we introduce a sliding window \(H_t\) to deal with these problems. Let \(H_t = \{(s_i, s_i', r_i) \mid j = \max(1, t - M + 1), \ldots, t\}\), where \(M\) is the window size. We only use the samples in \(H_t\) to evaluate \(h_t, g_t, p_t\), and \(q_t\); that is,
\[
\bar{h}_t = \sum_{j \in H_t} \Delta k_j(s_j, s_j'),
\]
\[
\bar{g}_t = \sum_{j \in H_t} k(s_j, s_j') \Delta k_{t-1}(s_t, s_t'),
\]
\[
\bar{p}_t = \sum_{j \in H_t} k(s_j, s_j) \Delta k_i(s_t, s_t') + \eta_t,
\]
\[
\bar{q}_t = \sum_{j \in H_t} k(s_j, s_j) r_j.
\] (23)

Then, similar to those in the first case, \(A_{t-1}\) and \(\alpha_t\) can be derived as follows:
\[
A_{t-1} = \frac{1}{m_{t-1}} \begin{bmatrix} m_t \bar{A}_{t-1} + \bar{A}_{t-1}^T \bar{h}_t \bar{g}_t \bar{A}_{t-1}^T - \bar{A}_{t-1}^T \bar{h}_t \\ -\bar{g}_t \bar{A}_{t-1} \end{bmatrix},
\] (24)
\[
\alpha_t = \frac{1}{m_t} \begin{bmatrix} m_t \bar{\alpha}_t - \bar{A}_{t-1}^T \bar{h}_t \bar{q}_t - \bar{g}_t \bar{\alpha}_t \\ \bar{q}_t - \bar{g}_t \bar{\alpha}_t \end{bmatrix},
\] (25)

where \(m_t = \bar{p}_t - \bar{g}_t \bar{A}_{t-1} \bar{h}_t\) and \(\bar{\alpha}_t\) is the same as the updated \(\alpha_t\) when the dictionary keeps unchanged.

Finally, we summarize the whole algorithm in Algorithm 1.

**Remark 1.** Here, we do not restrict the OSKRLSTD-\(L_2\) algorithm to a specific online sparsification method. That means it can be combined with many popular sparsification methods such as the novelty criterion (NC) [27] and the ALD criterion.

**Remark 2.** Although the OSKRLSTD-\(L_2\) algorithm is designed for infinite horizon tasks, it can be modified for episodic tasks. When \(s_t\) is an absorbing state, it only requires setting \(\gamma = 0\) temporarily and setting \(s_{t+1}\) as the start state of next episode.

**Remark 3.** Our simulation results show that a big sliding window cannot help improve the convergence performance of the OSKRLSTD-\(L_2\) algorithm. Thus, to save memory and reduce the computational cost, \(M\) should be set to a small integer.

### 3.2. OSKRLSTD-\(L_1\) Algorithm

In this subsection, we use \(L_1\) regularization and online sparsification to derive the second OSKRLSTD algorithm, which is called OSKRLSTD-\(L_1\).

First, we try to derive the \(L_1\)-regularized solution of (11). Add an \(L_1\)-norm penalty into (11); that is,
\[
\alpha_t = \tilde{f}(\alpha_t)
\]
\[
= \arg \min_{\beta \in \mathbb{R}^n} \left\| \widetilde{K}_t \beta - \left( \tilde{R}_t + \gamma(\tilde{K}_t)^T \alpha_t \right) \right\|_2 + 2\xi \left\| \beta \right\|_1,
\] (26)

where \(\xi \in [0, \infty)\) is a regularization parameter. However, \(\left\| \beta \right\|_1\) is not differentiable. Similar to Painter-Wakefield and Parr in [28], we resort to the subdifferential of \(g(\beta) = \left\| \widetilde{K}_t \beta - (\tilde{R}_t + \gamma(\tilde{K}_t)^T \alpha_t) \right\|_2 + 2\xi \left\| \beta \right\|_1\); that is,
\[
\nabla g(\beta) = 2\widetilde{K}_t \left( \tilde{R}_t + \gamma(\tilde{K}_t)^T \alpha_t \right) \right\|_2 + 2\xi \left\| \beta \right\|_1\),
\] (27)

where \(\xi \in [0, \infty)\) is a regularization parameter. However, \(\left\| \beta \right\|_1\) is not differentiable. Similar to Painter-Wakefield and Parr in [28], we resort to the subdifferential of \(g(\beta) = \left\| \widetilde{K}_t \beta - (\tilde{R}_t + \gamma(\tilde{K}_t)^T \alpha_t) \right\|_2 + 2\xi \left\| \beta \right\|_1\); that is,

\[\nabla g(\beta) = 2\widetilde{K}_t \left( \tilde{R}_t + \gamma(\tilde{K}_t)^T \alpha_t \right) \right\|_2 + 2\xi \left\| \beta \right\|_1\),
\] (27)

ALGORITHM 1: OSKRLSTD-\(L_2\)-
where \( \text{sgn}(\beta) \) is the set-valued function defined component-wise as
\[
\text{sgn}(\beta_j) = \begin{cases} 
{+1} & \beta_j > 0 \\
{-1,+1} & \beta_j = 0 \\
{-1} & \beta_j < 0.
\end{cases}
\] (28)

Let \( \forall g(\beta) = 0 \), so that
\[
\widehat{K}_t \left( K_t^\top \beta^* - \gamma \left( \widehat{K}_t^\top \alpha_t \right) \right) = \widehat{K}_t \widehat{R}_t - \xi \text{sgn}(\beta^*) .
\] (29)

Since \( w_t = u^* \), we also have \( \alpha_t = \beta^* \) from (9). Then, the above equation can be rewritten as
\[
\widehat{K}_t \left( K_t^\top \alpha_t \right) = \widehat{K}_t \widehat{R}_t - \xi \text{sgn}(\alpha_t) ,
\] (30)

where \( \text{sgn}(\alpha_t) \) has the same meaning as \( \text{sgn}(\beta) \). To avoid the singularity of \( \widehat{K}_t (K_t^\top - \gamma \widehat{K}_t^\top) \) and further reduce the complexity of the subsequent derivation, we introduce \( \eta \alpha_t \) into both sides; that is,
\[
\left( \widehat{K}_t \left( K_t^\top - \gamma \widehat{K}_t^\top \right) + \eta I_1 \right) \alpha_t = \widehat{K}_t \widehat{R}_t + \eta \alpha_t - \xi \text{sgn}(\alpha_t) ,
\] (31)

where \( \eta \in (0, \infty) \) is a regularization parameter. Obviously, the left hand side of (31) is the same as that of (14). Thus, from (16), the above equation can be rewritten as
\[
A_t \alpha_t = b_t + \eta \alpha_t - \xi \text{sgn}(\alpha_t) .
\] (32)

Then, we have the following fixed-point equation:
\[
\alpha_t = \mu_t + A_t^{-1} \left( \eta \alpha_t - \xi \text{sgn}(\alpha_t) \right) ,
\] (33)

where \( \eta \) denotes
\[
\mu_t = A_t^{-1} b_t .
\] (34)

Unfortunately, here, \( \alpha_t \) cannot be solved analytically.

Second, we investigate how to find the fixed point of (33). In \( L_1 \)-regularized LSTD algorithms [5, 29], researchers often used the LASSO method to tackle this problem. However, the LASSO method is inherently a batch method and is unsuitable for online learning. Instead, we resort to the fixed-point subiteration method introduced in [13]. We first use the sign function \( \text{sgn}(\alpha_t) \) to replace \( \text{sgn}(\alpha_t) \) in (33). Then, we can construct the following subiteration:
\[
\alpha_t^{l+1} = \mu_t + A_t^{-1} \left( \eta \alpha_t^l - \xi \text{sgn}(\alpha_t^l) \right) ,
\] (35)

where \( l \in \mathbb{N}^+ \) denotes the \( l \)-th subiteration and \( \alpha_t^1 \) is initialized to \( \mu_t \) since the fixed point will be close to \( \mu_t \) if \( \eta \) and \( \xi \) are small. If the subiteration number reaches a preset value \( N \in \mathbb{N}^+ \) or \( \| \alpha_t^{l+1} - \alpha_t^l \| \) is less than or equal to a preset threshold \( \epsilon \in \mathbb{R}^+ \), the subiteration will stop. From (32) and (28), if \( \| b_t + \eta \alpha_t - A_t \alpha_t \| < \xi, \alpha_{t,j} \) should be 0. Obviously, the replacement of \( \text{sgn}(\alpha_t) \) makes \( \alpha_t \) lose the ability to select features. To remedy this situation, after the whole subiteration, we remove the weakly dependent elements from \( \mathcal{D}_t \) according to the magnitude of \( \alpha_t \); that is,
\[
\mathcal{D}_t = \Psi_{\mathcal{J}_t}(\mathcal{D}_t) ,
\] (36)

where \( \Psi_{\mathcal{J}_t}(\cdot) \) denotes the operation to remove the elements indexed by the set \( \mathcal{J}_t \), which is determined by
\[
\mathcal{J}_t = \{ j \mid -\nu \leq \alpha_{t,j} \leq \nu, j = 1, \ldots, n_t - 1 \} ,
\] (37)

where \( \nu \in \mathbb{R}^+ \) is a preset threshold. Note that we do not remove the last element \( d_{n_t} \) of \( \mathcal{D}_t \), since \( |\alpha_{n_t}^l| \) is probably very small, especially when \( d_{n_t} \) is just added to \( \mathcal{D}_t \). Similarly, we perform \( \Psi_{\mathcal{J}_t}(\alpha_t) \) and \( \Psi_{\mathcal{J}_t}(\mu_t) \) to remove the weakly dependent coefficients. From (16), \( A_t^{-1} \) also requires removing some rows and columns. Unfortunately, we cannot use the method in [30] to do this like Chen et al. in [13], since \( A_t^{-1} \) is not a symmetry matrix. Considering that \( b_t \) will remove the corresponding elements if \( \mathcal{D}_t \) is pruned, we directly perform \( \Psi_{\mathcal{J}_t}(A_t^{-1}) \) to remove the rows and columns indexed by \( \mathcal{J}_t \). Although this method may bring some bias into \( A_t^{-1} \), our simulation results show that it is feasible and effective. The whole fixed-point subiteration and online pruning algorithm are summarized in Algorithm 2.

**Remark 4.** Our simulation results show that Algorithm 2 will converge in few iterations. Thus, Algorithm 2 does not become the computational bottleneck of the OSKRLSTD-L algorithm, and the maximum subiteration number \( N \) can be set to a small positive integer.

Third, we derive the recursive formulas of \( A_t^{-1} \) and \( \mu_t \). Although the dictionary can be pruned by using Algorithm 2, it still has the risk of rapidly growing if new samples are allowed to be added continually. Thus, the conventional sparsification method is also required to be considered here. Similar to Section 3.1, there are two cases under online sparsification. Since \( \Delta_t \) and \( \mu_t \) have the same definitions as \( \Delta_t \) and \( \mu_t \) in the OSKRLSTD-L algorithm, we can directly use (20) and (24) for updating \( A_t^{-1} \) and rewrite (21) and (25)
for updating $\mu_t$. If $s_t$ dissatisfies the sparsification condition, $\mu_t$ will be updated by

$$\mu_t = \mu_{t-1} + \frac{(r_t - \mu_{t-1}^T A_{t-1} k_{t-1} (s_t)) A_{t-1}^{-1} k_{t-1} (s_t)}{1 + \lambda \Delta k_{t-1}^T (s_t, s_t') A_{t-1}^{-1} k_{t-1} (s_t)}.$$  

(38)

Otherwise, $\mu_t$ will be updated by

$$\mu_t = \frac{1}{m_t} \left[ m_t \mu_t - \tilde{A}_{t-1}^{-1} \tilde{h}_t (\overline{q}_t - \tilde{h}_t^T \mu_t) \right],$$  

(39)

where $\tilde{h}_t$, $\tilde{g}_t$, $\overline{p}_t$, and $\overline{q}_t$ are also calculated by (23). Since $A_t$, $\tilde{A}_t$, and $\mu_t$ will be pruned by Algorithm 2 after the update, it is important to note that $\tilde{A}_t^{-1}$ and $\mu_t$ in (39) denote $A_t^{-1}$ and $\mu_t$ updated by $\mathcal{D}_{t-1}$ but unpruned by $\Psi_j$. Likewise, when (24) is used here, $\tilde{A}_t^{-1}$ has the same meaning.

Finally, we summarize the whole algorithm in Algorithm 3. For episodic tasks, the modification is the same as Remark 2. In addition, similar to Remark 3, the sliding-window size $M$ should also be set to a small integer.

Remark 5. By pruning the weakly dependent features, the OSKRLSTD-L$_1$ algorithm can yield a much sparser solution than the OSKRLSTD-L$_2$ algorithm.

4. Simulations

In this section, we use a nonnoise chain and a noise chain [2, 20, 31] to demonstrate the effectiveness of our proposed algorithms. For comparison purposes, RLSTD [1] and SKRLSTD [21] algorithms are also tested in the simulations.

To analyze the effect of regularization and online pruning on the performance of our algorithms, the OSKRLSTD-L$_2$ algorithm with $\eta = 0$ and the OSKRLSTD-L$_1$ algorithm with $\nu = 0$ (called OSKRLSTD-0 and OSKRLSTD-L$_1^u$, resp.) are tested here, too. In addition, the effects of the sliding-window size on the performance of our algorithms and OSKRLSTD-L$_1^u$ are evaluated as well.

4.1. Simulation Settings. As shown in Figure 1, in both chain problems, each chain consists of 50 states, which are numbered from 1 to 50. For each state, there are two actions available, that is, “left” (L) and “right” (R). Each action succeeds with probability 0.9, changing the state in the intended direction, and fails with probability 0.1, changing the state in the opposite direction. The two boundaries of each chain are dead-ends, and the discount factor $\gamma$ of each chain is set to 0.9. For the nonnoise chain, the reward is 1 only in states 10 and 41, whereas, for the noise chain, the reward is corrupted by an additive Gaussian noise $0.3 \mathcal{N}(0,1)$. Due to the symmetry, the optimal policy for both chains is to go right in states 1–9 and 26–41 and left in states 10–25 and 42–50. Here, we use it as the policy $\pi$ to be evaluated. Note that the state transition probabilities are available only for solving the true state-value functions $V^*$, and they are assumed to be unknown for all algorithms compared here.

In the implementations of all tested algorithms for both chain problems, the settings are summarized as follows: (i) For all OSKRLSTD algorithms, the Mercer kernel is defined as $k(x,y) = \exp(-\|x - y\|^2/16)$, the sparsification condition is defined as $\min_{d \in \mathcal{D}_{t-1}} \|s_t - d\| > 2$, and the sliding-window size $M$ is set to 5. Besides, for the OSKRLSTD-L$_1$ algorithm, the regularization parameters $\eta$ and $\xi$ are set to 0.8 and 0.3, the maximum subiteration number $N$ is set to 10, the precision threshold $\varepsilon$ is set to 0.1, and the pruning threshold $\nu$ is set to 0.4; for the OSKRLSTD-L$_1^u$ algorithm,
\( \eta, \xi, \) and \( N \) are the same as those in the \( \text{OSKRLSTD-} L_1 \) algorithm; for the \( \text{OSKRLSTD-} L_2 \) algorithm, \( \xi \) is set to 1. (ii) For the \( \text{SKRLSTD} \) algorithm, the Mercer kernel and the sparsification condition are the same as those in each \( \text{OSKRLSTD} \) algorithm. (iii) For the \( \text{RLSTD} \) algorithm, the feature vector \( \phi(s) \) consists of 19 Gaussian radius basis functions (GRBFs) plus a constant term 1, resulting in a total of 20 basis functions. The GRBF has the same definition as the Mercer kernel used in each \( \text{OSKRLSTD} \) algorithm, and the centers of GRBFs are uniformly distributed over \([1, 50]\). In addition, the variance matrix \( C_0 \) of \( \text{RLSTD} \) is initialized to \( 0.4I \), where \( I \) is the \( 20 \times 20 \) identity matrix. (iv) In the simulations, each algorithm performs 50 runs, each run includes 100 episodes, and each episode is truncated after 100 time steps. In particular, the \( \text{SKRLSTD} \) algorithm requires an extra run for offline sparsification before each regular run.

4.2. Simulation Results. We first report the comparison results of all tested algorithms with the simulation settings described in Section 4.1. Their learning curves are shown in Figure 2. At each episode, the root mean square error (RMSE) of each algorithm is calculated by RMSE = \((1/50) \sum_{j=1}^{50} (1/50) \sum_{s=1}^{50} (\hat{V}_j(s) - V^\pi(s))^2)^{0.5} \), where \( V^\pi(s) \) is solved by (1) and \( \hat{V}_j(s) \) is the approximate value of the \( j \)th run. From Figure 2, we can observe that (i) \( \text{OSKRLSTD-L}_2 \) and \( \text{OSKRLSTD-L}_1 \) can obtain the similar performance as \( \text{RLSTD} \) and converge much faster than \( \text{SKRLSTD} \). (ii) Without regularization, the performance of \( \text{OSKRLSTD-0} \) becomes very poor, especially in the noise chain. In contrast, \( \text{OSKRLSTD-L}_2 \) and \( \text{OSKRLSTD-L}_1 \) still perform well. (iii) The performance of \( \text{OSKRLSTD-L}_{1u} \) is only slightly better than that of \( \text{OSKRLSTD-L}_1 \), which indicates that online pruning has little effect on the performance. Figure 3 illustrates \( \hat{V}_j(s) \) approximated by all tested algorithms at the final episode. Clearly, \( \text{OSKRLSTD-0} \) has lost the ability to approximate \( V^\pi(s) \) of the noise chain. Figure 4 shows the dictionary growth curves of all tested algorithms. Compared with \( \text{RLSTD} \) and \( \text{SKRLSTD} \), all \( \text{OSKRLSTD} \) algorithms can construct the dictionary automatically, and \( \text{OSKRLSTD-L}_1 \) yields a much sparser dictionary. Figure 5 shows the average subiterations per time step in \( \text{OSKRLSTD-L}_1 \) and \( \text{OSKRLSTD-L}_{1u} \). As episodes increase, the subiterations decline gradually. In addition, online pruning can reduce the subiterations significantly. Even in the noise chain, the subiterations are small. Finally, the main simulation results of all tested algorithms at the final episode are summarized in Table 1.

Next, we evaluate the effect of the sliding-window size on our proposed algorithms and \( \text{OSKRLSTD-L}_{1u} \) with \( M = 1, 5, 10, \ldots, 45, 50 \). The logarithmic RMSEs of each algorithm at the final episode are illustrated in Figure 6. Note that the parameter settings of these algorithms are the same as those described in Section 4.1 except for \( M \). From Figure 6, \( \text{OSKRLSTD-L}_1 \) and \( \text{OSKRLSTD-L}_{1u} \) obviously become worse rather than better as the window size increases, and \( \text{OSKRLSTD-L}_2 \) has a strong adaptability to different window sizes. The reason for this result is analyzed as follows: From the derivation of our algorithms, the influence of the window size is mainly manifest in \( A_\eta^{-1} \). Since here \( A_\eta^{-1} \) is calculated by recursive update instead of matrix inversion and samples are used one by one, using too many history samples together may increase the calculation error. In \( \text{OSKRLSTD-L}_{2u} \), a moderate regularization parameter \( \eta \) can relieve the influence of this error. In contrast, in \( \text{OSKRLSTD-L}_1 \) and \( \text{OSKRLSTD-L}_{1u} \), the subiteration may expand the influence. Especially for \( \text{OSKRLSTD-L}_1 \), online pruning can introduce the new error, which further worsens the convergence performance. To verify the above analysis, we reset \( \eta = 0.6, \xi = 0.3, \) and \( N = 1 \) for \( \text{OSKRLSTD-L}_1 \) and \( \text{OSKRLSTD-L}_{1u} \) and reevaluate the effect of the window.
size. The new results are illustrated in Figure 7. As expected, OSKRLSTD-\(L_1\) and OSKRLSTD-\(L_{1u}\) can also adapt to \(M\). Nevertheless, there is still no proof that a big window size can help improve the convergence performance of OSKRLSTD-\(L_2\) and OSKRLSTD-\(L_1\). Thus, as stated in Remark 3, \(M\) is suggested to be set to a small integer in practice.

5. Conclusion

As an important approach for policy evaluation, LSTD algorithms can use samples more efficiently and eliminate all step-size parameters. But they require users to design the feature vector manually and often require many features to approximate state-value functions. Recently, there are some works on these issues by combining with sparse kernel methods. However, these works do not consider regularization and their sparsification processes are batch or offline. In this paper, we propose two online sparse kernel recursive least-squares TD algorithms with \(L_2\) and \(L_1\) regularization, that is, OSKRLSTD-\(L_2\) and OSKRLSTD-\(L_1\). By using Bellman operator along with projection operator, our derivation is more simple. By combining online sparsification, \(L_2\) and \(L_1\) regularization, recursive least squares, a sliding window, and the fixed-point subiteration, our algorithms not only can construct the feature dictionary online but also can avoid overfitting and eliminate the influence of noise. These
Table 1: Main simulation results on both chains at the final episode.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Nonnoise chain</th>
<th>Noise chain</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RMSE</td>
<td>Subiterations</td>
</tr>
<tr>
<td>RLSTD</td>
<td>0.47 ± 0.03</td>
<td>—</td>
</tr>
<tr>
<td>SKRLSTD</td>
<td>0.47 ± 0.05</td>
<td>15.36 ± 0.78</td>
</tr>
<tr>
<td>OKRLSTD-$L_2$</td>
<td>0.45 ± 0.05</td>
<td>15.30 ± 0.81</td>
</tr>
<tr>
<td>OKRLSTD-$L_1$</td>
<td>0.49 ± 0.08</td>
<td>11.52 ± 1.16</td>
</tr>
<tr>
<td>OKRLSTD-0</td>
<td>2.21 ± 0.05</td>
<td>15.25 ± 0.87</td>
</tr>
<tr>
<td>OKRLSTD-$L_{1u}$</td>
<td>0.44 ± 0.05</td>
<td>15.40 ± 0.76</td>
</tr>
</tbody>
</table>

(a) In the nonnoise chain

(b) In the noise chain

Figure 5: Average subiterations in OKRLSTD-$L_1$ and OKRLSTD-$L_{1u}$ algorithms.

Figure 6: Effect of the sliding-window size $M$ on three OKRLSTD algorithms.
advantages make them more suitable for online RL problems with a large or continuous state space. In particular, compared with the OSKRLSTD-\(L_2\) algorithm, the OSKRLSTD-\(L_1\) algorithm can yield a much sparser dictionary. Finally, we illustrate the performance of our algorithms and compare them with RLSTD and SKRLSTD algorithms by several simulations.

There are also some interesting topics to be studied in future work: (i) How to select proper regularization parameter should be investigated. (ii) A more thorough simulation analysis is needed, including an extension of our algorithms to learning control problems. (iii) Eligibility traces would be combined for further improving the performance of our algorithms. (iv) The convergence and prediction error bounds of our algorithms will be analyzed theoretically.

**Competing Interests**

The authors declare that there are no competing interests regarding the publication of this paper.

**Acknowledgments**

This work is supported in part by the National Natural Science Foundation of China under Grant nos. 61300192 and 11261015, the Fundamental Research Funds for the Central Universities under Grant no. ZYGX2014J052, and the Natural Science Foundation of Hainan Province, China, under Grant no. 613153.

**References**


Submit your manuscripts at
http://www.hindawi.com