Fast Plan with Linear Dyna - still in progress

Shuo YANG

Abstract—**In reinforcement learning problems, there is always a trade-off between data efficiency and computational efficiency. There are many different methods trying to achieve higher data efficiency by learning a model of the environment and planning, at the cost of heavier computational burden. In this paper, a new algorithm that have a $O(d)$ complexity in planning is presented, where d is the number of features in a linear function approximation. The new algorithm requires constant storage space and computation for every step in the Markov Decision Process (MDP), which makes it suitable for a large-scale, long-term learning. The new algorithm is based on the frame of linear Dyna architecture, with a different planning method. As demonstrated by the policy evaluation experiment on Boyan Chain domain and Mountain Car domain, the new algorithm could achieve the same data efficiency as the previous algorithms with much less computational cost. Experiments in Mountain Car domain further showed the new algorithm is also suitable for the control problem.**

I. Introduction

In reinforcement learning, the aim of the agent is to learn how to make optimal decisions in an unknown environment. The methods developed for solving RL problems can be divided into two main categories: model-free methods and model-based methods. Model-free methods don't need to learn a model to characterize the environment, all the information it requires to perform an update is the current observation, say new state and immediate reward. Like the most representative model-free method TD(0), it uses the current observation to calculate the TD error and update the corresponding state value or the value function, then the observation is discarded. Though the model-free methods could be very appealing due to its simplicity, only use the current observation to update the value function could lead to a very low data efficiency, which could become a problem when the cost of acquiring data is expensive.

As a contrast, the model-based method not only uses the current observation to learn a value function, the observation is also used to learn an internal model for the environment, such as the transition matrix and reward function of the environment. Though learning a model will definitely require more storage space and computational power, the advantage of a model is obvious: the agent could use the learned model to further updating its value function, without receiving new observation from the outside world. In this way, the agent is able to use the data more efficiently.

The trading off between data efficiency and computational efficiency often arises in the problem with a large state space or continuous state space. In those situations, another basic problem is that the tabular method can be infeasible, due to the large number of states. A commonly used and widely studied method is linear function approximation. In some environment, such as Boyan Chain [1], [2], the feature of a state would naturally be in the form of a vector which makes the linear function approximation quite convenient. There are still other domains where the state space is continuous. In those situations, we would need a method that could convert a continuous state representation to a linear vector representation. Tile coding presented by Sutton [12] is the method used in many papers on planning. To make the comparison clear, I also stick to tiling coding when dealing with the continuous domain.

By combining the model-based method with linear function approximation, we are starting constructing methods that could deal with a large environment with a high data efficiency. There are already works focus on this aspect, such as Linear Dyna [13]. Linera Dyna is an extension of the Dyna architecture [10] with linear function approximation, under the proof of the same convergence as the LSTD solution. In [10], different prioritized sweeping techniques [6]–[8] are also extended to the linear case, and a variation for control is also presented. However, due to the for loop in learning and planning, the time complexity of Linear Dyna is $O(d^2)$ which makes its computational efficiency unsatisfactory.

Another idea similar to linear Dyna would be LSTD method [3]. Calculating the inverse of a matrix would have a $O(d^3)$ complexity, but by using the information of the updating of a matrix, there is a method of $O(d^2)$ to compute the inverse. iLSTD [4], [5] shows that there could be a $O(d)$ method to achieve LSTD by iteration. iLSTD utilize the sparsity of feature vector to achieve linear time complexity. The idea behind iLSTD is very similar to Linear Dyna and it also converges to LSTD solution, which is same as Linear Dyna. However, the develop of iLSTD is trying to minimize the expectation of updating and always stick to one certain policy. Therefore it lacks the capability for control problem.

In this paper, I developed a new algorithm based on the Linear Dyna architecture. Instead of doing a prioritized sweeping, the linear equations were solved directly and iteratively in the planning part. Also by utilizing the sparsity in the feature representation, the time complexity of the algorithm is compacted to $O(d)$ without losing data efficiency compared with the previous method. Since the new algorithm doesn't require a queue to arrange sweeping, it only requires constant memory size for learning and planning, and the computation per time step remains constant as the observation grows. I further developed a variation for the control case, preserving the $O(d)$ time complexity and all other characteristics of the new algorithm.

II. Background

In this section, the notation adopted in this paper and other algorithms that are related to the new algorithm is presented.

A. Notation

For consistency, notation follows the standard framework for reinforcement learning with linear function approximation [12], which is also the notation adopted in the Linear Dyna by Sutton [13]. The series of observation and action performed by the agent is noted as a time indexed stream $s_0, a_0, r_1, s_1, r_2, s_2, ...,$ where $s_t \in \mathcal{S}$ is a state, $a_t \in \mathcal{A}$ is an action and $r_t \in \mathcal{R}$ is the immediate reward. The agent can only obtain the feature representation of a state $\boldsymbol{\phi}_t \in \mathcal{R}^n = \boldsymbol{\phi}(s_t)$. In a control problem, the feature will also take action into account $\boldsymbol{\phi}_t \in \mathcal{R}^n = \boldsymbol{\phi}(s_t, a_t)$. The agent selects action according to a policy, $\pi : \mathcal{R}^n \times \mathcal{A} \to [0, 1]$. By linear function approximation, the value function is approximated as a linear function with parameter vector $\boldsymbol{\theta} \in \mathcal{R}^n$:

$$V(s) = \boldsymbol{\theta}^\top \boldsymbol{\phi}(s) \approx V^\pi(s) = E_\pi \left\{ \sum_{t=1}^{\infty} \gamma^{t-1} r_t | s_0 = s \right\}$$

where $\gamma \in [0, 1)$. In this paper, the policies are greedy or ϵ-greedy with respect to the approximate state value function.

Throughout this paper, non-bolded symbols refer to scalars, bold-faced lower-case symbols refer to vectors, and bold-faced upper-case symbols refer to matrices.

B. TD(0) and Sarsa

Temporal difference (TD) learning [9] is the traditional approach for policy evaluation in reinforcement learning. TD learning can be guaranteed to converge with any linear function approximation and appropriate step-size setting [14]. The standard one-step TD method for value function approximation is TD(0). TD learning is also computationally in expensive, requiring only $O(n)$ computation per time step where n is the number of features used in approximation [4].

For the standard one-step TD method for value function approximation, TD(0)

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

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha_t \delta_t \boldsymbol{\phi}(s_t)$$

If only k features are non-zero for any state, then a sparse vector representation can further reduce the computation to $O(k)$ [4].

For a control problem, the agent has to learn an optimal policy through the interaction with the environment. Similar to TD(0) for state value, we can use Sarsa [12] to learn Q-Value and perform a greedy or ϵ-greedy policy. The learning process of Sarsa is listed below.

$$\delta_t = r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)$$

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha_t \delta_t \boldsymbol{\phi}(s_t, a_t)$$

where,

$$Q(s_t, a_t) = \boldsymbol{\theta}^\top \boldsymbol{\phi}(s_t, a_t)$$

C. Linear Dyna with prioritized sweeping

Sutton proposed linear Dyna architecture [13] and compared different prioritized sweeping method under linear function approximation. In Dyna architecture [10], we are learning a linear model of the environment, trying to predicting the next feature ϕ_{t+1} and reward r_{t+1} based on the current feature ϕ. So the learning process of the linear model would be

$$\begin{aligned} \boldsymbol{F}_{t+1} &= \boldsymbol{F}_t + \alpha(\boldsymbol{\phi}_{t+1} - \boldsymbol{F}_t \boldsymbol{\phi}_t)\boldsymbol{\phi}_t^\top \\ \boldsymbol{b}_{t+1} &= \boldsymbol{b}_t + \alpha(r_{t+1} - \boldsymbol{b}_t^\top \boldsymbol{\phi}_t)\boldsymbol{\phi}_t \end{aligned}$$

Based on the linear model of the environment F and b, the planning in linear Dyna would be

$$\begin{aligned} \boldsymbol{\phi}' &= \boldsymbol{F}\boldsymbol{\phi} \\ r &= \boldsymbol{b}^\top \boldsymbol{\phi} \\ \boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t + \alpha \left[r + \gamma \boldsymbol{\theta}_t^\top \boldsymbol{\phi}' - \boldsymbol{\theta}_t^\top \boldsymbol{\phi} \right] \boldsymbol{\phi} \end{aligned}$$

Algorithm 1 Linear Dyna with MG prioritized sweeping (policy evaluation)

```
1: Obtain initial $\boldsymbol{\phi}, \boldsymbol{\theta}, \boldsymbol{F}, \boldsymbol{b}$
2: for Each time step do
3:     Take action $a$ according to the policy. Receive $r, \boldsymbol{\phi}'$
4:     $\delta \leftarrow r + r\boldsymbol{\theta}^\top \boldsymbol{\phi}' - \boldsymbol{\theta}^\top \boldsymbol{\phi}$
5:     $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha\delta\boldsymbol{\phi}$
6:     $\boldsymbol{F} \leftarrow \boldsymbol{F} + \alpha(\boldsymbol{\phi}' - \boldsymbol{F}\boldsymbol{\phi})\boldsymbol{\phi}^\top$
7:     $\boldsymbol{b} \leftarrow \boldsymbol{b} + \alpha(r - \boldsymbol{b}^\top \boldsymbol{\phi})\boldsymbol{\phi}$
8:     for $i$ such that $\boldsymbol{\phi}(i) \neq 0$ do
9:         Put $i$ on the PQueue with priority $|\delta\boldsymbol{\phi}(i)|$
10:    for P times while PQueue is not empty do
11:        $i \leftarrow$ pop the PQueue
12:        for $j$ such that $\boldsymbol{F}^{ij} \neq 0$ do
13:            $\delta \leftarrow \boldsymbol{b}(j) + \gamma\boldsymbol{\theta}^\top \boldsymbol{F}\boldsymbol{e}_j - \boldsymbol{\theta}(j)$
14:            $\boldsymbol{\theta}(j) \leftarrow \boldsymbol{\theta}(j) + \alpha\delta$
15:            Put $j$ on the PQueue with priority $|\delta|$
16:    $\boldsymbol{\phi} \leftarrow \boldsymbol{\phi}'$
```

Algorithm 2 Linear Dyna with MG prioritized sweeping (control)

```
1: Obtain initial $\boldsymbol{\phi}, \boldsymbol{\theta}, \boldsymbol{F}, \boldsymbol{b}$
2: for Each time step do
3:     $a \leftarrow argmax_a[\boldsymbol{b}_a^\top \boldsymbol{\phi} + \gamma\boldsymbol{\theta}^\top \boldsymbol{F}_a \boldsymbol{\phi}]$ (or $\epsilon$-greedy)
4:     Take action $a$. Receive $r, \boldsymbol{\phi}'$
5:     $\delta \leftarrow r + r\boldsymbol{\theta}^\top \boldsymbol{\phi}' - \boldsymbol{\theta}^\top \boldsymbol{\phi}$
6:     $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha\delta\boldsymbol{\phi}$
7:     $\boldsymbol{F}_a \leftarrow \boldsymbol{F}_a + \alpha(\boldsymbol{\phi}' - \boldsymbol{F}_a\boldsymbol{\phi})\boldsymbol{\phi}^\top$
8:     $\boldsymbol{b}_a \leftarrow \boldsymbol{b}_a + \alpha(r - \boldsymbol{b}_a^\top \boldsymbol{\phi})\boldsymbol{\phi}$
9:     for $i$ such that $\boldsymbol{\phi}(i) \neq 0$ do
10:        Put $i$ on the PQueue with priority $|\delta\boldsymbol{\phi}(i)|$
11:    for P times while PQueue is not empty do
12:        $i \leftarrow$ pop the PQueue
13:        for $j$ s.t. there exists an $a$ s.t. $\boldsymbol{F}_a^{ij} \neq 0$ do
14:            $\delta \leftarrow max_a[\boldsymbol{b}_a(j) + \gamma\boldsymbol{\theta}^\top \boldsymbol{F}_a\boldsymbol{e}_j] - \boldsymbol{\theta}(j)$
15:            $\boldsymbol{\theta}(j) \leftarrow \boldsymbol{\theta}(j) + \alpha\delta$
16:            Put $j$ on the PQueue with priority $|\delta|$
17:    $\boldsymbol{\phi} \leftarrow \boldsymbol{\phi}'$
```

In the planning process, we are trying to minimize the TD(0) update based on the imaginary experience. That is

$$\mathbf{0} = \left(\boldsymbol{b} + \gamma\boldsymbol{F}^\top\boldsymbol{\theta} - \boldsymbol{\theta}\right)^\top \boldsymbol{\phi}$$

The only way that this can be true for all ϕ is for the expression in parenthesis above to be zero, which immediately implies that the planning of linear Dyna is

$$\boldsymbol{b} + (\boldsymbol{I} - \gamma\boldsymbol{F}^\top)\boldsymbol{\theta} = 0$$

As shown in the iLSTD paper [4], if $\boldsymbol{C} = \sum_{k=1}^{n} \boldsymbol{\phi}_k\boldsymbol{\phi}_k^\top$ has full rank, it has the same solution as LSTD.

Prioritized sweeping is designed for tabular Dyna, the idea behind is natural: try to propagating the value change to the states that lead to them. In prioritized sweeping, a prioritized queue will be maintained to keep track of the states and prioritized by its change in value. Thus the plan will always start from the state that may have the largest change in its value.

There are two prioritized sweeping algorithms used in the Linear Dyna paper. The first, coming from Peng and Williams [8] and also Moore and Atkeson [7] for tabular Dyna are called *PWMA prioritized sweeping*. The other one coming from McMahan and Gordon [6] are called *MG prioritized sweeping*. In Linear Dyna case, Dyna-MG is found to be the most efficient method in both policy evaluation and control [10]. Dyna-MG is used as the first baseline to compare with. The pseudocode of Dyna-MG for policy evaluation is shown in Algorithm 1, pseudocode for control is shown in Algorithm 2.

D. LSTD and iLSTD

The Least-Squares TD algorithm (LSTD) [3] can be seen as immediately solving for the value function parameters for which the sum TD update over all the observed data is zero. [4] Let $\mu_t(\theta)$ be the sum of the TD updates of all the observations through time t. For LSTD of TD(0):

$$\begin{aligned}
\boldsymbol{\mu}_t(\boldsymbol{\theta}) &= \sum_{i=1}^{t} \alpha\delta_t(V_{\boldsymbol{\theta}})\boldsymbol{\phi}(s_t) \\
&= \sum_{i=1}^{t} \boldsymbol{\phi}_t(r_{t+1} + \gamma\boldsymbol{\phi}_{t+1}^\top\boldsymbol{\theta} - \boldsymbol{\phi}_t^\top\boldsymbol{\theta}) \\
&= \sum_{i=1}^{t} (\boldsymbol{\phi}_t r_{t+1} - \boldsymbol{\phi}_t(\boldsymbol{\phi}_t - \gamma\boldsymbol{\phi}_{t+1})^\top\boldsymbol{\theta}) \\
&= \sum_{i=1}^{t} \boldsymbol{\phi}_t r_{t+1} - \sum_{i=1}^{t} \boldsymbol{\phi}_t(\boldsymbol{\phi}_t - \gamma\boldsymbol{\phi}_{t+1})^\top\boldsymbol{\theta} \\
&= (\boldsymbol{b}_t - \boldsymbol{A}_t\boldsymbol{\theta})
\end{aligned}$$

where,

$$\begin{aligned}
\boldsymbol{b}_t &= \sum_{i=1}^{t} \boldsymbol{\phi}_t r_{t+1} \\
\boldsymbol{A}_t &= \sum_{i=1}^{t} \boldsymbol{\phi}_t(\boldsymbol{\phi}_t - \gamma\boldsymbol{\phi}_{t+1})^\top
\end{aligned}$$

LSTD algorithm is trying to solve the last equation directly and update $\boldsymbol{\theta}$

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

In the non-incremental form, the matrix inversion alone would have a complexity of $O(d^3)$. Using the incremental form maintaining the matrix inversion would require $O(d^2)$ computation per time step.

In iLSTD [4], [5], another incremental computation for LSTD is presented. The main difference in iLSTD is updating $\boldsymbol{\theta}$ in an element-by-element fashion and update $\boldsymbol{\mu}$ vector by using the new $\boldsymbol{\theta}$. The relationship between $\boldsymbol{\mu}$ and $\Delta\boldsymbol{\theta}$ is given in the iLSTD paper

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

with a unit-basis vector ei, we have the iteration as follow

$$\begin{aligned} \boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t + \alpha_t\boldsymbol{\mu}_t(i)e_i \\ \boldsymbol{\mu}_t(\boldsymbol{\theta}_{t+1}) &= \boldsymbol{\mu}_t(\boldsymbol{\theta}_t) - \alpha_t\boldsymbol{\mu}_t(i)\boldsymbol{A}_t\boldsymbol{e}_i \end{aligned}$$

This iteration have the complexity of $O(d)$, and thus the complexity of iLSTD is $O(d)$.

The iteration of iLSTD is very similar to the planning method in the new linear Dyna algorithm that will be discussed in detail in next section. But they have two main difference. The first difference is equations solved in two methods. In LSTD and iLSTD, the learner matrix $\boldsymbol{A}$ and the vector $\boldsymbol{b}$ keep track of history TD update and trying to minimize it which makes it hard to generalize to a control problem. While in linear Dyna, the matrix $\boldsymbol{A}$ and the vector $\boldsymbol{b}$ is actually the model of the environment under a certain policy and therefore cable for a control problem where the policy may change as observation accumulates.The second difference is in the iteration. Although the iLSTD can do the iteration without a queue, it still relays on the trace $\boldsymbol{\mu}$ to guide for the element the would be updated, while in the new linear Dyna algorithm, the planning would iterate equally for all the elements in $\boldsymbol{\theta}$ using the Gauss-Siedel method without referring to a vector.

With the idea of calculating the inverse of a matrix to get the exact solution of the equations, I also apply it to the linear Dyna to develop another baseline to compare with. The equation in linear Dyna would be

$$(\gamma\boldsymbol{F}^\top - \boldsymbol{I})\boldsymbol{\theta} = \boldsymbol{b}$$

The pseudo code for this Direct-Solve algorithm is

Algorithm 3 Linear Dyna with Direct Solve (policy evaluation)

```
1: Obtain initial $\boldsymbol{\phi}, \boldsymbol{\theta}, \boldsymbol{F}, \boldsymbol{b}$
2: for Each time step do
3:     Take action $a$ according to the policy. Receive $r, \boldsymbol{\phi}'$
4:     $\boldsymbol{F} \leftarrow \boldsymbol{F} + \alpha(\boldsymbol{\phi}' - \boldsymbol{F}\boldsymbol{\phi})\boldsymbol{\phi}^\top$
5:     $\boldsymbol{b} \leftarrow \boldsymbol{b} + \alpha(r - \boldsymbol{b}^\top\boldsymbol{\phi})\boldsymbol{\phi}$
6:     $\boldsymbol{\theta} \leftarrow inverse(\boldsymbol{I} - \gamma\boldsymbol{F}^\top)\boldsymbol{b}$
```

III. New Algorithm

In this section, the new algorithm will be discussed in detail. The new algorithm mainly based on the linear Dyna architecture, with the planning part using the Gauss-Siedel method to reduce the time complexity and increase the converging speed. There are two different version of the new algorithm for policy evaluation and control.

A. Gauss-Siedel Method with relaxation

Gauss-Siedel method and its relaxation version is a commonly used method to solve linear equations iteratively. Consider a general set of linear equations:

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$$

we can decompose A by

$$\boldsymbol{A} = \boldsymbol{D} + \boldsymbol{A_L} + \boldsymbol{A_R}$$

where $\boldsymbol{D}$ is a diagonal matrix, $\boldsymbol{A_L}$ is a lower left triangular matrix, $\boldsymbol{A_R}$ is an upper right triangular matrix.

Then we can rewrite the linear equations into

$$(\boldsymbol{D} + \boldsymbol{A_L})\boldsymbol{x} = -\boldsymbol{A_R}\boldsymbol{x} + \boldsymbol{y}$$

$$\boldsymbol{x} = -(\boldsymbol{D} + \boldsymbol{A_L})^{-1}\boldsymbol{A_R}\boldsymbol{x} + (\boldsymbol{D} + \boldsymbol{A_L})^{-1}\boldsymbol{y}$$

which is then solved by the successive approximations

$$\boldsymbol{x}_{v+1} = -(\boldsymbol{D}+\boldsymbol{A_L})^{-1}\boldsymbol{A_R}\boldsymbol{x}_v + (\boldsymbol{D}+\boldsymbol{A_L})^{-1}\boldsymbol{y}, v = 0, 1, 2...$$

This is the Gauss-Siedel method in matrix formation. We can also gain the Gauss-Siedel iteration element-wise from the matrix equation.

$$\boldsymbol{x}_{v+1,j} = -\sum_{k=1}^{j-1}\frac{a_{jk}}{a_{jj}}\boldsymbol{x}_{v+1,k} - \sum_{k=j+1}^{n}\frac{a_{jk}}{a_{jj}}\boldsymbol{x}_{v,k} + \frac{\boldsymbol{y}_j}{a_{jj}} \qquad j = 1, ..., n.$$

The Gauss-Siedel iteration can also be written as

$$\boldsymbol{x}_{v+1} = \boldsymbol{x}_v + \boldsymbol{D}^{-1}[\boldsymbol{y} - \boldsymbol{A_L}\boldsymbol{x}_{v+1} - (\boldsymbol{D} + \boldsymbol{A_R})\boldsymbol{x}_v]$$

If we have a relaxation factor ω for the error term, them we get

$$\boldsymbol{x}_{v+1} = \boldsymbol{x}_v + \omega \boldsymbol{D}^{-1}[\boldsymbol{y} - \boldsymbol{A_L}\boldsymbol{x}_{v+1} - (\boldsymbol{D} + \boldsymbol{A_R})\boldsymbol{x}_v]$$

The iteration with a relaxation factor ω also have a element wise formation

$$\boldsymbol{x}_{v+1,j} = \boldsymbol{x}_{v,j} + \frac{\omega}{a_{jj}}\left[\boldsymbol{y}_j - \sum_{k=1}^{j-1} a_{jk}\boldsymbol{x}_{v+1,k} - \sum_{k=j}^{n} a_{jk}\boldsymbol{x}_{v,k}\right] \quad j = 1, ..., n.$$

This iteration is known as Gauss-Siedel method with relaxation with *relaxation coefficient* $\omega > 0$. As we can easily see from the equation, each iteration only change one element of the unknown vector $\boldsymbol{x}$ and has a complexity of $O(d)$ where d is the number of elements in vector $\boldsymbol{x}$. Also, since the updating is in-place, the updating only requires $O(1)$ extra storage space.

It has been shown that the necessary condition for the iteration to converge is $0 < \omega < 2$. When $\omega > 1$ the method will be called *Over-Relaxation*. The *relexation coefficient* ω controls the spectral radius of the iteration matrix

$$\boldsymbol{B}(\omega) = (\boldsymbol{D} + \omega\boldsymbol{A_L})^{-1}[(1-\omega)\boldsymbol{D} - \omega\boldsymbol{A_R}]$$

The spectral radius controls the convergence and the convergent speed of the iteration. Since $\boldsymbol{B}(\omega)$ is non-linearly dependent on ω, there is not a simple way to find an optimal ω or even decide whether there exist a ω that can make the iteration converge. The current way is to try ω in experiment.

B. Algorithm for Policy Evaluation

In the problem of policy evaluation, we here only consider the situation for on-policy evaluation. Though incorporate techniques as importance sampling could lead to a off-policy version of linear Dyna, the off-policy learning is beyond the scope of this paper.

For a fixed policy to be evaluated, we can always learn a single set of F and b to characterize the transition of the state and its corresponding reward, as has been discussed in the background section. After we gained an estimation of F and b, we can use the Gauss-Siedel method to iteratively solve the linear equations

$$(\gamma\boldsymbol{F}^\top - \boldsymbol{I})\boldsymbol{\theta} = \boldsymbol{b}$$

In this way, we no longer have to keep a prioritized queue to serve for the planning. Every element in θ will be updated with equal frequency.The pseudocode is shown below.

Algorithm 4 Linear Dyna with G-S method in planning (policy evaluation)

```
1: Obtain initial $\boldsymbol{\phi}, \boldsymbol{\theta}, \boldsymbol{F}, \boldsymbol{b}, \boldsymbol{m} \leftarrow \boldsymbol{0}$
2: for Each time step do
3:     Take action $a$ according to the policy. Receive $r, \boldsymbol{\phi}'$
4:     $\delta \leftarrow r + r\boldsymbol{\theta}^\top\boldsymbol{\phi}' - \boldsymbol{\theta}^\top\boldsymbol{\phi}$    ▷ $O(k)$
5:     $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha\delta\boldsymbol{\phi}$    ▷ $O(k)$
6:     $\boldsymbol{F} \leftarrow \boldsymbol{F} + \alpha(\boldsymbol{\phi}' - \boldsymbol{F}\boldsymbol{\phi})\boldsymbol{\phi}^\top$    ▷ $O(d)$
7:     $\boldsymbol{b} \leftarrow \boldsymbol{b} + \alpha(r - \boldsymbol{b}^\top\boldsymbol{\phi})\boldsymbol{\phi}$    ▷ $O(k)$
8:     for p times do
9:         $\boldsymbol{C} = \boldsymbol{I} - \gamma\boldsymbol{F}^\top$
10:        $\delta \leftarrow \frac{1}{\boldsymbol{C}_{mm}}\left(\boldsymbol{b}_m - \boldsymbol{C}_{:,m}^\top\boldsymbol{\theta}\right)$    ▷ $O(d)$
11:        $\boldsymbol{\theta}(m) \leftarrow (1-\omega)\boldsymbol{\theta}(m) + \omega\delta$    ▷ $O(1)$
12:        $m \leftarrow m + 1$    ▷ $O(1)$
13:        if $m > d$ then
14:            $m \leftarrow 1$    ▷ $O(1)$
15:    $\boldsymbol{\phi} \leftarrow \boldsymbol{\phi}'$
```

If there are d features and for any given state s, $\boldsymbol{\phi}(s)$ has at most k non-zero elements, which is usually the case, especially when using tile coding, then the time complexity for each line is commented on the right side in the pseudo code. As we can see, each plan step will have a linear time complexity and thus the algorithm has linear time complexity, under the assumption of $k \ll d$

C. Algorithm for control

The learning process for control situation is similar to the one in Dyna-MG. Since we don't have a fixed policy to generate action, we need to learn the transition matrix $\boldsymbol{F}$ and $\boldsymbol{b}$ condition on each possible action a. Therefore the learning $\boldsymbol{F}$ would become the learning of corresponding $\boldsymbol{F}_a$

However, there will be still one more problem for the new algorithm applying to control. Because the selection of action a is dependent on $\boldsymbol{F}_a, \boldsymbol{b}, \boldsymbol{\theta}$, the planning part cannot be simply seen as solving the linear equations we discussed in the policy evaluation situation.

To address this problem, we learn a separate $\boldsymbol{F}$ and $\boldsymbol{b}$, which is not dependent on the action. In this way, the learned $\boldsymbol{F}$ and $\boldsymbol{b}$ are characterizing the transition under the current policy. In the planning part, we can then use the Gauss-Siedel method to solve the linear equations defined by $\boldsymbol{F}$ and $\boldsymbol{b}$. This is analogous to the policy evaluation although the target policy is changing accordingly to $\boldsymbol{F}_a, \boldsymbol{b}_a, \boldsymbol{\theta}$. The pseudo code for control is shown as algorithm 5.

Algorithm 5 Linear Dyna with G-S method in planning (Control)

```
1: Obtain initial $\boldsymbol{\phi}, \boldsymbol{\theta}, \boldsymbol{F}, \boldsymbol{b}, \boldsymbol{m} \leftarrow \boldsymbol{0}$
2: for Each time step do
3:     $a \leftarrow argmax_a[\boldsymbol{b}_a^\top \boldsymbol{\phi} + \gamma \boldsymbol{\theta}^\top \boldsymbol{F}_a \boldsymbol{\phi}]$ (or $\epsilon$-greedy)
4:     Take action $a$. Receive $r, \boldsymbol{\phi}'$
5:     $\delta \leftarrow r + r\boldsymbol{\theta}^\top \boldsymbol{\phi}' - \boldsymbol{\theta}^\top \boldsymbol{\phi}$        ▷ $O(k)$
6:     $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha\delta\boldsymbol{\phi}$        ▷ $O(k)$
7:     $\boldsymbol{F}_a \leftarrow \boldsymbol{F}_a + \alpha(\boldsymbol{\phi}' - \boldsymbol{F}_a\boldsymbol{\phi})\boldsymbol{\phi}^\top$        ▷ $O(d)$
8:     $\boldsymbol{b}_a \leftarrow \boldsymbol{b}_a + \alpha(r - \boldsymbol{b}_a^\top \boldsymbol{\phi})\boldsymbol{\phi}$        ▷ $O(k)$
9:     $\boldsymbol{F} \leftarrow \boldsymbol{F} + \alpha(\boldsymbol{\phi}' - \boldsymbol{F}\boldsymbol{\phi})\boldsymbol{\phi}^\top$        ▷ $O(d)$
10:    $\boldsymbol{b} \leftarrow \boldsymbol{b} + \alpha(r - \boldsymbol{b}^\top \boldsymbol{\phi})\boldsymbol{\phi}$        ▷ $O(k)$
11:    for p times do
12:        $\boldsymbol{C} = \boldsymbol{I} - \gamma \boldsymbol{F}^\top$
13:        $\delta \leftarrow \frac{1}{\boldsymbol{C}_{mm}}\left(\boldsymbol{b}_m - \boldsymbol{C}_{:,m}^\top \boldsymbol{\theta}\right)$        ▷ $O(d)$
14:        $\boldsymbol{\theta}(m) \leftarrow (1-\omega)\boldsymbol{\theta}(m) + \omega\delta$        ▷ $O(1)$
15:        $m \leftarrow m + 1$        ▷ $O(1)$
16:        if $m > d$ then
17:            $m \leftarrow 1$        ▷ $O(1)$
18:    $\boldsymbol{\phi} \leftarrow \boldsymbol{\phi}'$
```

As the discussion for policy evaluation, still assume $k \ll d$, then the time complexity for each line is commented on the right side of the pseudo code. As we can see, the algorithm also has the complexity of $O(d)$. The only difference between control and policy evaluation is the number of $\boldsymbol{F}$ and $\boldsymbol{b}$ that has to be updated during the learning process, which won't affect the computational efficiency too much due to the linear time complexity.

IV. Experiment Result

In this section, we will first introduce the two experimental domain: Boyan chain, and mountain car environment. Policy evaluation experiments are carried out on both Boyan chain and mountain car domain, and run the control experiment on the mountain car domain. The experiment detail and result are discussed in detail in each subsection. In all the experiment result, the curve labeled with TD(0) and Sarsa are using the standard TD(0) and Sarsa algorithm, the curve labeled with "Dyna-MG" are using the algorithm 1, 2, labeled with "Direct Solve" are using algorithm 3, labeled with "Over Relaxation" are using algorithm 4, 5.

A. Experiment Domain

1) Boyan Chain: The Boyan Chain environment is an extension of the one by [1], [2] with variable environment length and feature number. Figure 1 shows the

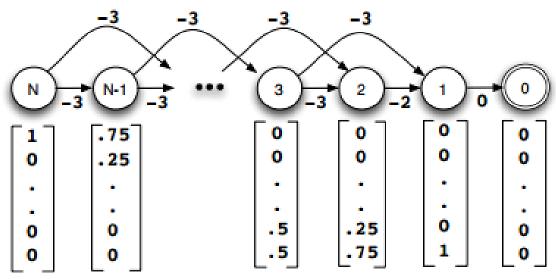

Figure 1. Boyan Chain Environment

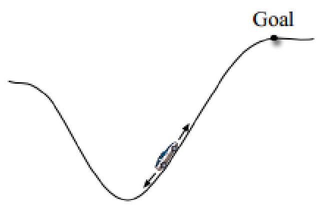

Figure 2. Mountain Car Environment

environment in a general form. Each episode starts at state with maximum N and terminates in state 0. For all state $s > 2$, there is an equal probability of transitioning to state $s-1$ or $s-2$ with a reward of -3. From states 2 and 1, there are deterministic transitions to states 1 and 0 with respective rewards of -2 a 0. This variation comes from the Boyan Chain used in the linear Dyna [13].

In the empirical experiment, we can set the number of features to an arbitrary integer to get a Boyan Chain of a different length. The length of the environment is roughly 4 times the length of the feature. Besides we can easily get a more complex environment, the Boyan Chain domain also have another advantage that the real state value can be obtained analytically easily, which make the calculation of value error simple. Since the transition is fixed and not dependent on the agent's action, this environment is only for policy evaluation experiment use.

2) Mountain Car: The Mountain Car domain is the same as the one described by [11], [12]. An underpowered car must be driven to the right top of a hill by rocking back and forth in a valley. The setting of the environment is same as the one described in the linear Dyna paper [13]. The state variables are a pair (position, velocity) initialized to (-0.5, 0.0) at the beginning of each episode. The reward is -1 per time step. There are three discrete actions (accelerate, reverse, and coast). The continuous state space is converted to vector presentation by tile coding with 10 tilings over the combined (position, velocity) pair, and with the tiles

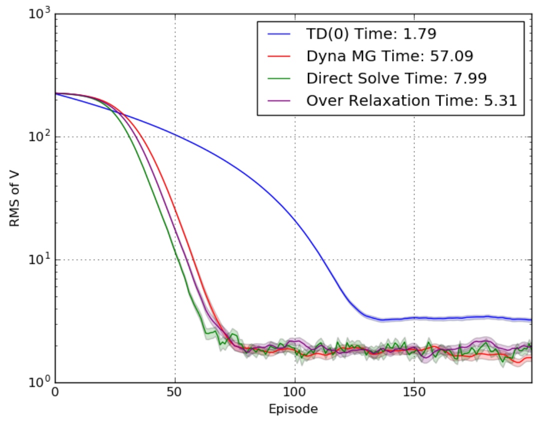

Figure 3. Boyan Chain of 50 features

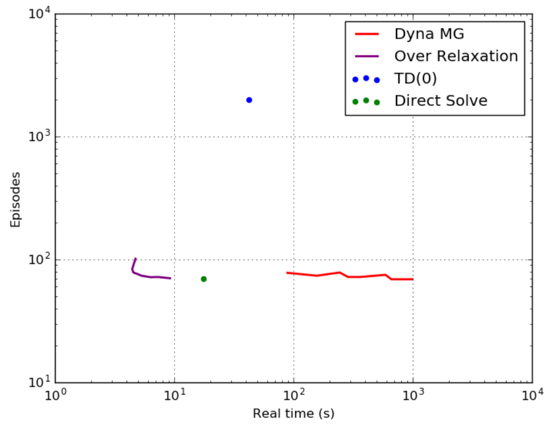

Figure 4. Episodes and time to converge in Boyan Chain of 100 features

hashed down to 1,000 features. This environment is used both in policy evaluation and control problem. In all the experiment, the policy has $\epsilon = 0.1$ and $\gamma = 1$.

B. Policy Evaluation

1) Boyan Chain: In the policy evaluation experiment, the change of MSVE by different training methods as the training episodes grow is first evaluated. The number of features was set to 50 in this experiment. The TD(0) learner has $\alpha = 0.5$ while all other learners have $\alpha = 0.05$. The Dyna-MG learner will plan for 1 time in each time step, while the Over Relaxation learner will plan for 3 times per time step. The relaxation coefficient ω is set to 0.5 for all the following experiment including the control one. The experiment is repeated for 30 times, the standard error is the shaded area on the figure. The running time of each repeat is attached to the label.

As we can see, the methods that learn a model of the environment converge faster than the model-free TD(0) learner as expected, even with a smaller step size α. The three method roughly have the same data efficiency as suggested by the convergence speed. But the real time consumed varies a lot, with the new algorithm 10 times faster than the previous Dyna-MG method.

Note that although Over Relaxation learner is allowed to plan 3 times more than Dyna-MG, because of the different mechanism of planning, 1 step plan in Dyna-MG will lead into almost every element in θ to be updated, 1 step plan in Over Relaxation only allow one element in θ to change. Therefore the updating of Over Relaxation is significantly less than Dyna-MG, which leads to a higher computational efficiency.

Because the motivation of the new algorithm is the tradeoff between data efficiency and computational efficiency, an experiment compare them directly is carried out. In this experiment, the length of features is increased to 100. The step size of the different learner is set to same at $\alpha = 0.05$. The experiment will let the agent interact with the environment until the learning converges, say have the MSVE less than 5. The total episode number and real-time before converge will be recorded to compare the data efficiency and computational efficiency. For the learner with planning, the plan step will be set to all the values in $[1, 2, 3, 4, 5, 8, 10, 15]$ to give a thorough estimation of the algorithm. The result is as follow.

As shown in figure 4, the learner with modeling will have similar data efficiency, but the new algorithm will be generally faster than Dyna-MG by an order. It's notable that although TD(0) requires the least computational time per step, due to the low data efficiency, the real-time it consumed to converge is longer than the new algorithm.

The final experiment with Boyan Chain domain will exam the increasing speed of episodes to converge and real-time to converge as the environment become more complex. Using the same setting for the agents as in the previous experiment, except the planning step for both Dyna-MG and Over Relaxation is fixed to 3. The number of features of the environment is varied as $[5, 10, 15, 25, 50, 80, 100]$. The corresponding change of episodes to converge and time to converge is shown in figure 5 and figure 6.

The three model-based methods will have almost the same data efficiency as the number of features grow. But for the time consuming, Direct Solve will grow faster than the other two due to the $O(d^3)$ complexity, while the new algorithm is always more than an order faster

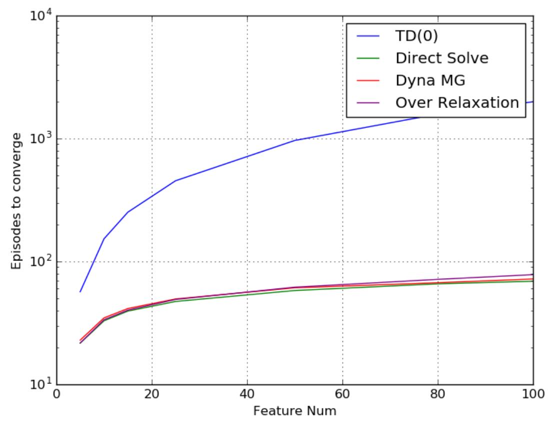

Figure 5. Episodes to converge in Boyan Chain with increasing features

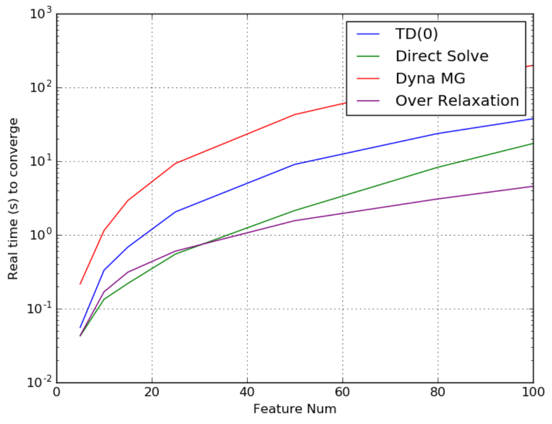

Figure 6. Time to converge in Boyan Chain with increasing features

than Dyna-MG in this scope.

2) Mountain Car: In the Mountain Car domain, the policy to be evaluated is the same as the one shown in Linear Dyna Paper [13]. The policy was to accelerate in the direction of the current velocity, with the noise that switched the selected action to a random action with 10% probability.

For the configuration of the learners, we set TD(0) with $\alpha = 0.03$ and other learners with $\alpha = 0.01$, which is the optimal setting for them. The planning step of Dyna-MG and Over Relaxation was set to 1 and 15. The experiment will be run for 1000 episodes, with each 100 episode the parameter will be frozen to get an estimation of current policy evaluation. Each estimation will be run for 5000 episodes and calculate the mean square error. Since the state value can't be obtained analytically as in Boyan Chain domain. Instead, the TD error will be used

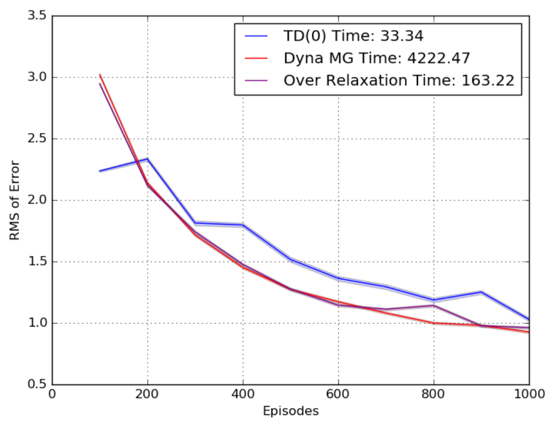

Figure 7. Policy evaluation in Mountain Car Environment

for estimation, which is similar to Sutton's experiment [13]. The result is shown in figure 7, and the real-time consumed in learning is attached to the label. Due to the complexity of the environment, Direct Solve learner is not included here.

As the curves show, the model based learner has a similar better data efficiency. But the time used by the Over Relaxation learner is 163 seconds while Dyna-MG used 4222 seconds, more than 25 times slower.

C. Control

The new algorithm is also tested as a control algorithm in the Mountain Car domain, comparing with Dyna-MG as well as the model-free Sarsa. The setting of the environment, the configuration of tile coding, are kept same as in policy evaluation experiment.

For the configuration of the learners, all the step sizes were set to 0.01. The planning step of Dyna-MG and Over Relaxation is set to 1 and 15. The experiment runs for 80 episodes with 3 repeats. The standard error is the shaded area in the figure. The steps in an episode are shown in figure 8, and the per step time is shown in figure 9.

As we can see, the Dyna-MG and Over Relaxation also has the same data efficiency in the control problem, the steps per episode converge at almost same speed and significantly faster than Sarsa. But their time per step has a big difference, with the Dyna-MG almost slower than Over Relaxation by two orders.

There is still another notable point. In figure 9, we can see an increase at the beginning of the curve of Dyna-MG, this is because as the matrix F being filled with non-zero elements, the planning speed will be slower.

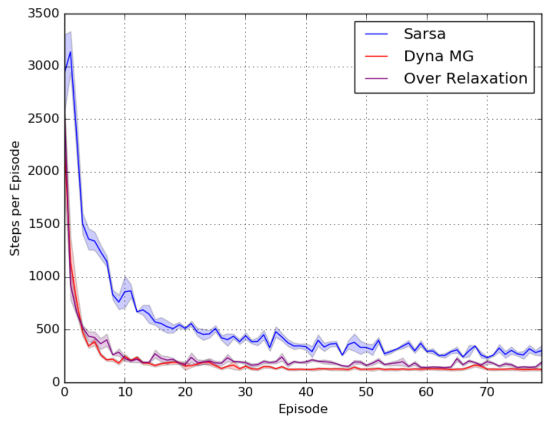

Figure 8. Steps per episode in Mountain Car control problem

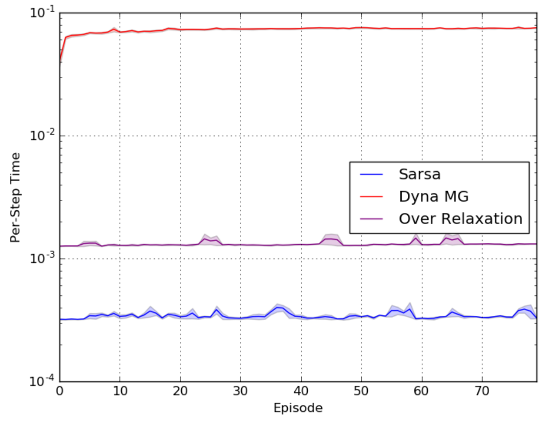

Figure 9. Time per step in Mountain Car control problem

Also, as the size of the priority queue in Dyna-MG keep growing as the observation accumulates, the enqueue operation will be more an more time consuming and the queue itself will require a larger and larger storage space. This makes Dyna-MG can't really deal with the long-term learning, since the computational cost and required storage space will keep growing. A method partial solves this problem would be limiting the size of the priority queue. Since the new algorithm doesn't require a queue, it won't be affected by this problem and will have a constant per-time-step time-consuming.

V. Discussion

In this paper, a new model-based reinforcement learning algorithm is presented based on the linear Dyna architecture. The new algorithm can achieve the same data efficiency as the previous linear Dyna method with $O(d)$ complexity. The per time step computation of the new algorithm is constant as the experience accumulates, which makes it both scalable and suitable for long-term learning. The experiment shows that the new algorithm is able to deal with both policy evaluation and control problem with high data efficiency and computational efficiency.

However, there are still problems with the new algorithm. Although the experiments have demonstrated the great performance of the new algorithm, since there is no theoretical guarantee for the convergence of using Gauss-Siedel iteration with relaxation, there is the potential risk that the algorithm may diverge in some environment. As observed in the Boyan Chain experiment, if $\omega > 0.5$ the iteration will actually diverge. So the existence of ω that could ensure the convergence of iteration, or the calculation of the optimal ω is still in question.

In addition, there is also other possible future work based on this new algorithm. The current work is for the on-policy problem, but with importance sampling or other techniques, this work could be extended to the off-policy problem. Also, the idea behind this new algorithm is to solve the linear equations described by the learned model directly and iteratively. So besides Gauss-Siedel method, there are still other methods to be explored, which can leard to a more stable and faster convergence.

References

[1] Boyan, J. A. Least-squares temporal difference learning. In *ICML* (1999), pp. 49–56.

[2] Boyan, J. A. Technical update: Least-squares temporal difference learning. *Machine Learning 49*, 2 (2002), 233–246.

[3] Bradtke, S. J., and Barto, A. G. Linear least-squares algorithms for temporal difference learning. In *Recent Advances in Reinforcement Learning*. Springer, 1996, pp. 33–57.

[4] Geramifard, A., Bowling, M., and Sutton, R. S. Incremental least-squares temporal difference learning. In *Proceedings of the National Conference on Artificial Intelligence* (2006), vol. 21, Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, p. 356.

[5] Geramifard, A., Bowling, M., Zinkevich, M., and Sutton, R. S. ilstd: Eligibility traces and convergence analysis. In *Advances in Neural Information Processing Systems* (2007), pp. 441–448.

[6] McMahan, H. B., and Gordon, G. J. Fast exact planning in markov decision processes. In *ICAPS* (2005), pp. 151–160.

[7] Moore, A. W., and Atkeson, C. G. Prioritized sweeping: Reinforcement learning with less data and less time. *Machine learning 13*, 1 (1993), 103–130.

[8] Peng, J., and Williams, R. J. Efficient learning and planning within the dyna framework. *Adaptive Behavior 1*, 4 (1993), 437–454.

[9] Sutton, R. S. Learning to predict by the methods of temporal differences. *Machine learning 3*, 1 (1988), 9–44.

[10] Sutton, R. S. Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *Proceedings of the seventh international conference on machine learning* (1990), pp. 216–224.

[11] Sutton, R. S. Generalization in reinforcement learning: Successful examples using sparse coarse coding. In *Advances in neural information processing systems* (1996), pp. 1038–1044.

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

[13] Sutton, R. S., Szepesvári, C., Geramifard, A., and Bowling, M. P. Dyna-style planning with linear function approximation and prioritized sweeping. *arXiv preprint arXiv:1206.3285* (2012).

[14] Tsitsiklis, J. N., and Van Roy, B. Analysis of temporal-diffference learning with function approximation. In *Advances in neural information processing systems* (1997), pp. 1075–1081.