A Deeper Look at Planning as Learning from Replay

Harm van Seijen HARM.VANSEIJEN@UALBERTA.CA
Richard S. Sutton SUTTON@CS.UALBERTA.CA
Department of Computing Science, University of Alberta, Edmonton, Alberta, T6G 2E8, Canada

Abstract

In reinforcement learning, the notions of experience replay, and of planning as learning from replayed experience, have long been used to find good policies with minimal training data. Replay can be seen either as model-based reinforcement learning, where the store of past experiences serves as the model, or as a way to avoid a conventional model of the environment altogether. In this paper, we look more deeply at how replay blurs the line between model-based and model-free methods. First, we show for the first time an exact equivalence between the sequence of value functions found by a model-based policy-evaluation method and by a model-free method with replay. Second, we present a general replay method that can mimic a spectrum of methods ranging from the explicitly model-free (TD(0)) to the explicitly model-based (linear Dyna). Finally, we use insights gained from these relationships to design a new model-based reinforcement learning algorithm for linear function approximation. This method, which we call *forgetful LSTD(λ)*, improves upon regular LSTD(λ) because it extends more naturally to online control, and improves upon linear Dyna because it is a multi-step method, enabling it to perform well even in non-Markov problems or, equivalently, in problems with significant function approximation.

1. Introduction

In reinforcement learning (RL) (Sutton & Barto, 1998; Kaelbling et al., 1996; Szepesvári, 2009), value-function based methods are traditionally divided into two categories: model-free and model-based. Model-free, or direct, methods learn a value function directly from samples, from which a policy can be easily derived. By contrast, model-based, or indirect, methods learn a model of the environment dynamics and derive a value function or policy from the model using planning.

The distinction between model-free and model-based (and similarly, learning and planning) is not always clear. For example, with linear Dyna (Sutton et al., 2008) samples are used to simultaneously update a value function and learn a model. The model is then used to generate simulated samples to further improve the value function. Another example are model-free methods that store observed samples for reuse at a later time, like LSTD (Bradtke & Barto, 1996; Boyan, 2002) or experience replay (Lin, 1992; Adam et al., 2012). While such methods do not store an explicit transition model, they share many characteristics with model-based methods, such as increased computational requirements and improved sample efficiency, which has led some people to argue that a stored set of samples should also be considered a model.

The distinction between model-free and model-based methods is further blurred by the theoretical result that the fixed point of a linear, least-squares model on a set of samples is the same as the LSTD solution of those samples (Sutton et al., 2008; Parr et al., 2008). In practise, however, the value function computed by a model-based method (at any particular time) is different. This is partly because it learns only an estimate of the least-squares model, and because, due to bounded computation per time step, its iterative planning process terminates before convergence is reached.

In this paper, we show that the relation between model-free and model-based methods runs even deeper. Specifically, we show for the first time an exact equivalence between the sequence of value functions found by a model-based method and by a model-free method with replay.

The replay technique we use is different from the traditional way of doing replay as introduced by Lin (1992). Rather than presenting old samples as new to the learning agent, we recompute the update targets of old samples based on current information and redo the updates start-

Proceedings of the 32^{nd} International Conference on Machine Learning, Lille, France, 2015. JMLR: W&CP volume 37.

ing from the initial estimate. This has the advantage that the order of samples remains intact and that it can be combined with any update target, including the frequently used λ-return (Sutton, 1988). Traditional replay prohibits this because it interleaves new samples with old samples. We present a general algorithm based on this replay technique that can mimic a whole spectrum of other methods ranging from model-free methods, like TD(0), to model-based methods using a linear model, to anything in between.

Using a specific instantiation of our general replay method, we derive a new method that can be interpreted both as a variation on LSTD(λ) that gracefully forgets old information and as a model-based method based on a multi-step linear model. Inspired by the former interpretation, we call this method *forgetful LSTD(λ)*. Forgetting old information is important in control tasks, where the policy changes over time. On the other hand, multi-step models are key for obtaining robust model-based behaviour in problems with significant function approximation, or similarly, state aggregation. This is not widely known, however, as evidenced by a number of recent publications that conclude that learned models appear to be fundamentally limited, based on observed behaviour from a one-step model. We illustrate the importance of multi-step models with a small experiment.

This paper is organized as follows. After introducing the problem setting, we present our replay technique and show that combining it with TD(0) is equivalent to planning based on a learned linear model. We then present a general replay algorithm that can mimic a whole range of model-free and model-based methods. Finally, based on a specific instantiation of this general replay algorithm, we derive forgetful LSTD(λ).

2. Problem Setting and Notation

In this section, we present the main learning framework. In addition, we define the learning function, which will play a key role in this paper. As a convention, we indicate random variables by capital letters (e.g., S_t, R_t), vectors by bold letters (e.g., $\boldsymbol{\theta}$, $\boldsymbol{\phi}$), functions by lowercase letters (e.g., v), and sets by calligraphic font (e.g., $\mathcal{S}$, $\mathcal{A}$).

2.1. Markov Reward Processes

We focus in this paper primarily on discrete-time *Markov reward processes* (MRPs), which can be described as 4-tuples of the form $\langle \mathcal{S}, p, r, \gamma \rangle$, consisting of $\mathcal{S}$, the set of all states; $p(s'|s)$, the transition probability function, giving for each state $s \in \mathcal{S}$ the probability of a transition to state $s' \in \mathcal{S}$ at the next step; $r(s, s')$, the reward function, giving the expected reward after a transition from s to s'. γ is the discount factor, specifying how future rewards are weighted with respect to the immediate reward.

The *return* at time step t is the discounted sum of rewards observed after time step t:

$$G_t = \sum_{i=1}^{\infty} \gamma^{i-1} R_{t+i} \,.$$

The value-function v of an MRP maps each state $s \in \mathcal{S}$ to the expected value of the return:

$$v(s) = \mathbb{E}\{G_t \,|\, S_t = s\} \,.$$

We consider the case where the learner does not have access to s directly, but can only observe a feature vector $\boldsymbol{\phi}(s) \in \mathbb{R}^n$. We estimate the value function using linear function approximation, in which case the value of a state is the inner product between the weight vector $\boldsymbol{\theta}$ and a feature vector $\boldsymbol{\phi}$. In this case, the value of state s at time step t is approximated by:

$$\hat{v}_t(s) = \boldsymbol{\theta}_t^\top \boldsymbol{\phi}(s) \,.$$

As a shorthand, we will indicate $\boldsymbol{\phi}(S_t)$, the feature vector of the state visited at time step t, by $\boldsymbol{\phi}_t$.

A general model-free update rule for linear function approximation —inspired by stochastic gradient descent— is:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha \, [U_t - \boldsymbol{\theta}_t^\top \boldsymbol{\phi}_t] \boldsymbol{\phi}_t \,, \tag{1}$$

where U_t, the *update target*, is some estimate of the expected return at time step t. There are many ways to construct an update target. For example, the TD(0) update target is, indicated by $G_t^{(1)}$, is:

$$G_t^{(1)} = R_{t+1} + \gamma \boldsymbol{\theta}_t^\top \boldsymbol{\phi}_{t+1} \,.$$

A generalization of this update target is the n*-step return*, which is based on a feature vector observed n steps later in time.

$$G_t^{(n)} = \sum_{i=1}^{n} \gamma^{i-1} R_{t+i} + \gamma^n \, \boldsymbol{\theta}_t^\top \boldsymbol{\phi}_{t+n} \,.$$

The λ-return generalizes this further by taking a weighted sum over n-step returns:

$$G_t^\lambda = (1 - \lambda) \sum_{n=1}^{\infty} \lambda^{n-1} G_t^{(n)} \,.$$

Note that for $\lambda = 0$, the λ-return reduces to the TD(0) update target, while for $\lambda = 1$ it reduces to the full return.

It is often useful to extend the basic MRP definition with the concept of *terminal states*. The interpretation of a terminal state is that the return for the current *episode* is terminated and the state is reset to the initial state (possibly drawn from a fixed distribution), which starts a new episode.

The concept of terminal states can be modelled using a state-dependent discount factor $\gamma(s)$, that is equal to 0 if the state s is a terminal state. Because we consider the scenario where the learner does not have direct access to states, we model terminal states using a time-dependent γ that is produced by the environment, and observed by the learner in addition to the reward and feature vector. When a terminal state is reached, the learner is instantly transported to the initial state and γ is set to 0. This way of modelling episodic tasks makes that we can treat episodic tasks and continuing tasks fully uniformly. Hence, the algorithms we introduce will not explicitly consider episodes; they consider time-dependent γ instead.

2.2. The Learning Function

The quality of a prediction algorithm, be it a model-free or a model-based method, is determined by two distinct aspects: its computational complexity (in terms of space and time) and the weight vectors it computes. In this paper, we formalize the latter by the *learning function*, h, that outputs a weight vector, given a sample sequence $\Omega_t = \{\boldsymbol{\phi}_0, R_1, \boldsymbol{\phi}_1, \ldots, R_t, \boldsymbol{\phi}_t\}$ and a set of internal parameters. For example, for TD(0) the learning function looks like:

$$\boldsymbol{\theta}_t = h(\Omega_t | \alpha, \boldsymbol{\theta}_{init}) \,,$$

where α is the step-size parameter and $\boldsymbol{\theta}_{init}$ is the initial value of the weight vector.

This definition allows us to talk about equivalence of algorithms from the perspective of the value function. Algorithms can have very different computational complexities, but still have the same learning function.

This notion of equivalence has been around for a long time. For example, it underlies the forward/backward view of eligibility traces (Sutton & Barto, 1998): the forward view provides a conceptually simple learning method; while the backward view provides an efficient implementation.

3. New Interpretation of Planning

In RL, the act of planning is typically thought of as looking ahead. For example, the one-step linear model used by the Dyna architecture gives, for an arbitrarily input feature vector, the expected next feature vector and reward. Planning is thought of as exploiting this ability by performing an update in the direction of the expected future reward and state value. We introduce an alternative viewpoint that, rather than interpreting planning as looking ahead, interprets it as a re-evaluation of the past using current information. In particular, we demonstrate that the one-step linear model of Dyna can be interpreted both ways.

3.1. Planning by Re-Evaluating the Past

In this section, we illustrate the basic idea of re-evaluating the past and we introduce a method based on this principle.

Consider a learning agent that performs regular TD(0) updates at the first three time steps, starting from the initial weight vector $\boldsymbol{\theta}_0$:

$$\begin{aligned}
\boldsymbol{\theta}_1 &= \boldsymbol{\theta}_0 + \alpha \left[R_1 + \gamma \boldsymbol{\theta}_0^\top \boldsymbol{\phi}_1 - \boldsymbol{\theta}_0^\top \boldsymbol{\phi}_0\right] \boldsymbol{\phi}_0 \\
\boldsymbol{\theta}_2 &= \boldsymbol{\theta}_1 + \alpha \left[R_2 + \gamma \boldsymbol{\theta}_1^\top \boldsymbol{\phi}_2 - \boldsymbol{\theta}_1^\top \boldsymbol{\phi}_1\right] \boldsymbol{\phi}_1 \\
\boldsymbol{\theta}_3 &= \boldsymbol{\theta}_2 + \alpha \left[R_3 + \gamma \boldsymbol{\theta}_2^\top \boldsymbol{\phi}_3 - \boldsymbol{\theta}_2^\top \boldsymbol{\phi}_2\right] \boldsymbol{\phi}_2 \,.
\end{aligned}$$

Now consider that after performing these three updates, the agent takes a moment to reflect upon the updates it made. It would realize that the update targets — based on $\boldsymbol{\theta}_0^\top \boldsymbol{\phi}_1, \boldsymbol{\theta}_1^\top \boldsymbol{\phi}_2$ and $\boldsymbol{\theta}_2^\top \boldsymbol{\phi}_3$ — all use outdated weight vectors with respect to its current weight vector $\boldsymbol{\theta}_3$. Subsequently, it could reason about how its current weight vector would look like, if it had used $\boldsymbol{\theta}_3$ from the start to construct the update targets:

$$\begin{aligned}
\boldsymbol{\theta}'_1 &= \boldsymbol{\theta}_0 + \alpha \left[R_1 + \gamma \boldsymbol{\theta}_3^\top \boldsymbol{\phi}_1 - \boldsymbol{\theta}_0^\top \boldsymbol{\phi}_0\right] \boldsymbol{\phi}_0 \\
\boldsymbol{\theta}'_2 &= \boldsymbol{\theta}'_1 + \alpha \left[R_2 + \gamma \boldsymbol{\theta}_3^\top \boldsymbol{\phi}_2 - (\boldsymbol{\theta}'_1)^\top \boldsymbol{\phi}_1\right] \boldsymbol{\phi}_1 \\
\boldsymbol{\theta}'_3 &= \boldsymbol{\theta}'_2 + \alpha \left[R_3 + \gamma \boldsymbol{\theta}_3^\top \boldsymbol{\phi}_3 - (\boldsymbol{\theta}'_2)^\top \boldsymbol{\phi}_2\right] \boldsymbol{\phi}_2 \,.
\end{aligned}$$

It seems reasonable to assume that $\boldsymbol{\theta}'_3$ is more accurate than $\boldsymbol{\theta}_3$, because it is constructed from update targets that use a more recent weight vector. By that same reasoning, an even better weight vector $\boldsymbol{\theta}''_3$ can be obtained, by replaying the updates once more, now with update targets based on $\boldsymbol{\theta}'_3$ instead of $\boldsymbol{\theta}_3$. This process can be repeated until the weight vector converges or the budget of computation time has ran out. Algorithm 1 shows an algorithm that performs this form of re-evaluation at every time step.

Algorithm 1 Replaying TD(0) updates

INPUT: $\alpha, k, \boldsymbol{\theta}_{init}$
$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta}_{init}$
$\mathcal{V} \leftarrow \emptyset$ // *$\mathcal{V}$ is a list of observed samples*
obtain initial $\boldsymbol{\phi}$
Loop:
  obtain next feature vector $\boldsymbol{\phi}'$, γ and reward R
  add $(\boldsymbol{\phi}, R, \gamma, \boldsymbol{\phi}')$ to $\mathcal{V}$
  Repeat k times:
    $\boldsymbol{\theta}_* \leftarrow \boldsymbol{\theta}$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta}_{init}$
    For all $(\dot{\boldsymbol{\phi}}, \dot{R}, \dot{\gamma}, \dot{\boldsymbol{\phi}}')$ in $\mathcal{V}$ (from oldest to newest):
      $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \left[\dot{R} + \dot{\gamma}\, \boldsymbol{\theta}_*^\top \dot{\boldsymbol{\phi}}' - \boldsymbol{\theta}^\top \dot{\boldsymbol{\phi}}\right] \dot{\boldsymbol{\phi}}$
  $\boldsymbol{\phi} \leftarrow \boldsymbol{\phi}'$

Note that this form of replaying experience is different from the traditional way, introduced by Lin (1992). In the traditional way, previously observed samples are presented

to the learning agent as new samples. By contrast, in Algorithm 1, the replay is started from the initial weight vector and the weight vector used for the update targets is fixed. One advantage of this approach is that it can be applied to any type of update target, including multi-step update targets such as the λ-return. Traditional experience replay cannot be combined with the λ-return (Sutton, 1988), because the λ-return requires an online sample sequence and mixing old samples with new samples breaks this condition.

Obviously, Algorithm 1 is very inefficient when long sample sequences are involved: both memory and computation per time step increases linearly with the number of observed samples. But we can now search for a more efficient implementation with the same learning function. This leads to a surprising result, as we show in the next section.

3.2. Relation to Linear Models

While on an abstract level, planning by looking ahead and planning by re-evaluating the past appear to be opposite approaches, there is a surprising relation when applied to RL. We demonstrate this by rewriting the updates of Algorithm 1 in terms of vectors and matrices.

Let $\{\boldsymbol{\phi}_0, R_1, \gamma_1, \boldsymbol{\phi}_1, \ldots, R_t, \gamma_t \boldsymbol{\phi}_t\}$ be the observed experience sequence at time t. In addition, let $\boldsymbol{\theta}_{(t)}$ be the result of a single replay sweep over the t samples in $\mathcal{V}$. Then, $\boldsymbol{\theta}_{(t)}$ is incrementally defined by:

$$\boldsymbol{\theta}_{(i+1)} = \boldsymbol{\theta}_{(i)} + \alpha \left[R_{i+1} + \gamma \boldsymbol{\theta}_*^\top \boldsymbol{\phi}_{i+1} - \boldsymbol{\theta}_{(i)}^\top \boldsymbol{\phi}_i\right] \boldsymbol{\phi}_i \,,$$

for $0 \leq i < t$ and with $\boldsymbol{\theta}_{(0)} := \boldsymbol{\theta}_{init}$. This update can be rewritten as:

$$\boldsymbol{\theta}_{(i+1)} = (\mathcal{I} - \alpha \boldsymbol{\phi}_i \boldsymbol{\phi}_i^\top)\, \boldsymbol{\theta}_{(i)} + \alpha \boldsymbol{\phi}_i \Big[R_{i+1} + \gamma \boldsymbol{\phi}_{i+1}^\top \boldsymbol{\theta}_*\Big] \,, \quad (2)$$

where $\mathcal{I}$ is the identity matrix. Using (2), $\boldsymbol{\theta}_{(1)}$ can be written as:

$$\begin{aligned} \boldsymbol{\theta}_{(1)} &= (\mathcal{I} - \alpha \boldsymbol{\phi}_0 \boldsymbol{\phi}_0^\top) \boldsymbol{\theta}_{(0)} + \alpha \boldsymbol{\phi}_0 R_1 + \alpha\gamma \boldsymbol{\phi}_0 \boldsymbol{\phi}_1^\top \boldsymbol{\theta}_* \\ &= \boldsymbol{b}_1 + F_1^\top \boldsymbol{\theta}_* \,, \end{aligned} \quad (3)$$

where $\boldsymbol{b}_1$ is a vector of size n and $F_1^\top$ is an $n \times n$ matrix (the transpose of a matrix F_1—which we discuss later), defined as:

$$\begin{aligned} \boldsymbol{b}_1 &:= (\mathcal{I} - \alpha \boldsymbol{\phi}_0 \boldsymbol{\phi}_0^\top) \boldsymbol{\theta}_{(0)} + \alpha \boldsymbol{\phi}_0 R_1 \\ F_1^\top &:= \alpha\gamma \boldsymbol{\phi}_0 \boldsymbol{\phi}_1^\top \,. \end{aligned}$$

By writing down the update for $\boldsymbol{\theta}_{(2)}$ and substituting (3), the following is obtained:

$$\begin{aligned} \boldsymbol{\theta}_{(2)} &= (\mathcal{I} - \alpha \boldsymbol{\phi}_1 \boldsymbol{\phi}_1^\top) \boldsymbol{\theta}_{(1)} + \alpha \boldsymbol{\phi}_1 \Big[R_2 + \gamma \boldsymbol{\phi}_2^\top \boldsymbol{\theta}_*\Big] \\ &= (\mathcal{I} - \alpha \boldsymbol{\phi}_1 \boldsymbol{\phi}_1^\top) \boldsymbol{b}_1 + (\mathcal{I} - \alpha \boldsymbol{\phi}_1 \boldsymbol{\phi}_1^\top) F_1^\top \boldsymbol{\theta}_* + \\ &\quad + \alpha \boldsymbol{\phi}_1 R_2 + \alpha\gamma \boldsymbol{\phi}_1 \boldsymbol{\phi}_2^\top \boldsymbol{\theta}_* \Big] \\ &= \boldsymbol{b}_2 + F_2^\top \boldsymbol{\theta}_* \,, \end{aligned}$$

with

$$\begin{aligned} \boldsymbol{b}_2 &:= (\mathcal{I} - \alpha \boldsymbol{\phi}_1 \boldsymbol{\phi}_1^\top) \boldsymbol{b}_1 + \alpha \boldsymbol{\phi}_1 R_2 \\ F_2^\top &:= (\mathcal{I} - \alpha \boldsymbol{\phi}_1 \boldsymbol{\phi}_1^\top) F_1^\top + \alpha\gamma \boldsymbol{\phi}_1 \boldsymbol{\phi}_2^\top \,. \end{aligned}$$

By repeating this process up to time t, it follows that a single iteration of the update sequence of Algorithm 1 can be written as:

$$\boldsymbol{\theta}_{(t)} = \boldsymbol{b}_t + F_t^\top \boldsymbol{\theta}_* \,, \quad (4)$$

where $F_t^\top$ and $\boldsymbol{b}_t$ are incrementally defined by:

$$\begin{aligned} \boldsymbol{b}_{i+1} &:= (\mathcal{I} - \alpha \boldsymbol{\phi}_i \boldsymbol{\phi}_i^\top) \boldsymbol{b}_i + \alpha \boldsymbol{\phi}_i R_{i+1} & (5) \\ F_{i+1}^\top &:= (\mathcal{I} - \alpha \boldsymbol{\phi}_i \boldsymbol{\phi}_i^\top) F_i^\top + \alpha\gamma \boldsymbol{\phi}_i \boldsymbol{\phi}_{i+1}^\top \,. & (6) \end{aligned}$$

for $1 \leq i < t$. Note that by initializing $\boldsymbol{b}_0 = \boldsymbol{\theta}_{init}$ and $F_0^\top = \mathbf{0}$ (that is, $F_0^\top$ is equal to an all-zero matrix), $\boldsymbol{b}_1$ and $F_1^\top$ can also be computed using these equations.

Equation (5) can be rewritten as:

$$\boldsymbol{b}_{i+1} = \boldsymbol{b}_i + \alpha \big[R_{i+1} - \boldsymbol{b}_i^\top \boldsymbol{\phi}_i\big] \boldsymbol{\phi}_i \,. \quad (7)$$

In addition, Equation (6) can be rewritten as an update for F, resulting in:

$$\begin{aligned} F_{i+1} &= \Big[(\mathcal{I} - \alpha \boldsymbol{\phi}_i \boldsymbol{\phi}_i^\top) F_i^\top + \alpha\gamma \boldsymbol{\phi}_i \boldsymbol{\phi}_{i+1}^\top\Big]^\top \\ &= F_i (\mathcal{I} - \alpha \boldsymbol{\phi}_i \boldsymbol{\phi}_i^\top)^\top + \alpha\gamma \boldsymbol{\phi}_{i+1} \boldsymbol{\phi}_i^\top \\ &= F_i (\mathcal{I} - \alpha \boldsymbol{\phi}_i \boldsymbol{\phi}_i^\top) + \alpha\gamma \boldsymbol{\phi}_{i+1} \boldsymbol{\phi}_i^\top \\ &= F_i + \alpha \big[\gamma \boldsymbol{\phi}_{i+1} - F_i \boldsymbol{\phi}_i\big] \boldsymbol{\phi}_i^\top \,. \end{aligned} \quad (8)$$

Using equations (4), (7) and (8) Algorithm 2 can be constructed, which has the same learning function as Algorithm 1, but a different computational complexity. Specifically, the memory and computation per time step do *not* increase with the number of observed samples.

The surprising part about Algorithm 2 is that it uses the same one-step model, and model updates, as the Dyna architecture (Sutton et al., 2008), which is based on the concept of planning by looking ahead. This demonstrates that planning by re-evaluating the past or planning by looking ahead are just two different ways of looking at the same phenomena.

Algorithm 2 Planning with the linear Dyna model

INPUT: $\alpha, k, \boldsymbol{\theta}_{init}$
$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta}_{init}$ // *vector of size* n
$\boldsymbol{b} \leftarrow \boldsymbol{\theta}_{init}$ // *vector of size* n
$F \leftarrow \mathbf{0}$ // $n \times n$ *matrix*
obtain initial $\boldsymbol{\phi}$
Loop:
  obtain next feature vector $\boldsymbol{\phi}'$, γ and reward R
  $F \leftarrow F + \alpha\big[\gamma\boldsymbol{\phi}' - F\boldsymbol{\phi}\big]\boldsymbol{\phi}^\top$
  $\boldsymbol{b} \leftarrow \boldsymbol{b} + \alpha(R - \boldsymbol{b}^\top\boldsymbol{\phi})\boldsymbol{\phi}$
  Repeat k times:
    $\boldsymbol{\theta} \leftarrow \boldsymbol{b} + F^\top\boldsymbol{\theta}$
  $\boldsymbol{\phi} \leftarrow \boldsymbol{\phi}'$

4. General Planning by Replay

In this section, we generalize the replay algorithm introduced in the previous section. In particular, we extend it to arbitrary update targets. We also extend it to retain previous estimates to various degrees. This enables it to mimic a whole spectrum of methods ranging from the simplest model-free method, TD(0), to full model-based methods.

4.1. Generalized Algorithm

Algorithm 3 shows pseudo-code of a general method based on planning by replay. It contains three sub-methods: *replay*, *compute_targets* and *update_weights*.

The *replay* method takes as input a list of update targets and a list of feature vectors, and computes a weight vector by sequentially performing updates, starting from the initial weight vector. Algorithm 4 shows an implementation for use with linear function approximation.

The *compute_targets* method computes a list of update targets. As input it uses a list of ⟨reward, discount factor, feature vector⟩ tuples and a list of weight vectors. Each weight vector corresponds with exactly one sample. Algorithm 5 shows an example implementation that computes 1-step returns. However, any type of update target is possible. In the next section, we discuss the case where interim λ-returns are computed.

The *update_weights* method updates the weight vectors used to construct the update targets. At time t, the agent has visited t states whose value can be used in an update target. The list $\mathcal{W}$ assigns to each of these visited states a weight vector that will be used to determine its value. The update_weights method sets the weight vectors in $\mathcal{W}$ corresponding to the m most recent states equal to the current weight vector.

The value of m determines whether the learning function of Algorithm 3 behaves like that of a model-free or model-based method — or something in between. More specifically, for $m = 1$ and $k = 1$ (and compute_targets as in Algorithm 5), the learning function of Algorithm 3 is equal to that of TD(0). On the other hand, for $m = \infty$, the learning function is equal to that of Algorithm 2 (and of Algorithm 1). Using a value for m in between these two extremes will result in learning functions with behaviour in between that of TD(0) and a one-step model. Other weight update strategies are also possible, but updating the weights for the most recent states makes sense, because, due to the steps-size, the values of older states have less impact on the current weight vector.

Algorithm 3 General Planning by Replay

INPUT: $\boldsymbol{\theta}_{init}, \alpha, k, m$ // $k \in \mathbb{N}^+$, $m \in \mathbb{N}^+$
$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta}_{init}$
$\Phi, \mathcal{Y}, \mathcal{W}, \leftarrow \emptyset$ // $\Phi, \mathcal{Y}$ *and* $\mathcal{W}$ *are lists*
obtain initial $\boldsymbol{\phi}$
Loop:
  obtain next feature vector $\boldsymbol{\phi}'$, γ and reward R
  add $\boldsymbol{\phi}$ to Φ; add $\langle R, \gamma, \boldsymbol{\phi}'\rangle$ to $\mathcal{Y}$; add $\boldsymbol{\theta}$ to W
  Repeat k times:
    $\mathcal{W} \leftarrow update\_weights(\mathcal{W}, \boldsymbol{\theta}, m)$
    $\mathcal{U} \leftarrow compute\_targets(\mathcal{Y}, W)$
    $\boldsymbol{\theta} \leftarrow replay(\Phi, \mathcal{U}, \alpha, \boldsymbol{\theta}_{init})$
  $\boldsymbol{\phi} \leftarrow \boldsymbol{\phi}'$

Algorithm 4 replay (invoked at time t)

INPUT: $\Phi, \mathcal{U}, \alpha, \boldsymbol{\theta}_{init}$
OUTPUT: $\boldsymbol{\theta}$
// $\Phi = \langle \boldsymbol{\phi}_1, \ldots, \boldsymbol{\phi}_t \rangle$ *: list of feature vectors*
// $\mathcal{U} = \langle U_1, \ldots, U_t \rangle$ *: list of update targets*

$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta}_{init}$
For $i = 1 : t$
  $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha(U_i - \boldsymbol{\theta}^\top\boldsymbol{\phi}_i)\boldsymbol{\phi}_i$

Algorithm 5 compute_targets (invoked at time t)

INPUT: $\mathcal{Y}, \mathcal{W}$
OUTPUT: U
// $\mathcal{Y} = \langle\langle R_1, \gamma_1, \boldsymbol{\phi}'_1\rangle, \ldots, \langle R_t, \gamma_t, \boldsymbol{\phi}'_t\rangle\rangle$
// $\mathcal{W} = \langle \boldsymbol{w}_1, \ldots, \boldsymbol{w}_t \rangle$*: list of weight vectors*
// $\mathcal{U} = \langle U_1, \ldots, U_t \rangle$*: list of update targets*

For $i = 1 : t$
  $U_i \leftarrow R_i + \gamma_i \boldsymbol{w}_i^\top \boldsymbol{\phi}'_i,$

4.2. The Interim λ-return

The general advantage of replay is that the update target of a visited state can be recomputed using new information that was not available just after the state was visited. So

Algorithm 6 update_weights (invoked at time t)

INPUT: $\mathcal{W}, \boldsymbol{\theta}, m$
OUTPUT: $\mathcal{W}'$
// $\mathcal{W} = (\boldsymbol{w}_1, \ldots, \boldsymbol{w}_t)$*: list of weight vectors*
// $\boldsymbol{\theta}$ *: current weight vector*
// $m \in \mathbb{N}^+$
// $\mathcal{W}' = (\boldsymbol{w}'_1, \ldots, \boldsymbol{w}'_t)$*: list of (updated) weight vectors*

For $i = 1 : t$
  if $i > t - m$:
    $\boldsymbol{w}'_i \leftarrow \boldsymbol{\theta}$
  else :
    $\boldsymbol{w}'_i \leftarrow \boldsymbol{w}_i$

far, we just considered 1-step update targets and focussed on using more accurate weight vectors in replay. Another option is to use new samples to construct a more sophisticated update target. For example, an increasing n-step return could be used, where n is kept equal to the difference between the current time step and the time step the relevant state was visited.

The *interim λ-return* generalizes the idea of an increasing n-step return. It was recently introduced to derive an improved version of TD(λ), called true online TD(λ) (van Seijen & Sutton, 2014). The interim λ-return can be interpreted as a finite version of the regular λ-return. It uses at most an n-step return in its weighted sum, with n equal to the time difference between the current time step and the time step the state for which the update target is computed was visited.

To compute interim λ-returns, the update of U_i in Algorithm 5 should be changed to:

$$U_i \leftarrow (1-\lambda) \sum_{n=1}^{t-i-1} \lambda^{n-1} G_i^{(n)} + \lambda^{t-i-1} G_i^{(t-i)} ,$$

with

$$G_i := \sum_{j=1}^{n} R_{i+j} \prod_{k=1}^{j-1} \gamma_{i+k} + \boldsymbol{\theta}_{i+n}^\top \boldsymbol{\phi}_{i+n} \prod_{k=1}^{n} \gamma_{i+k} .$$

With this update target and with $k = 1$ and $m = 1$, the learning function of Algorithm 3 is equal to that of true online TD(λ).

5. Forgetful LSTD(λ)

In this section, we introduce a new model-based algorithm that combines a multi-step model, similar to the one of LSTD(λ), with graceful forgetting, as occurs in linear Dyna.

5.1. Derivation

When interim λ-returns are used as update targets and m is set to ∞, Algorithm 3 mimics a model-based version of true online TD(λ). Rewriting the updates of this method in terms of vectors and matrices (just as was done for Algorithm 1 in Section 3.2) results in the following updates:

$$\boldsymbol{b}_{i+1} = \big(\mathcal{I} - \alpha \boldsymbol{\phi}_i \boldsymbol{\phi}_i^\top\big) \boldsymbol{b}_i + \alpha \boldsymbol{e}_{i+1} R_{i+1} \tag{9}$$

$$F_{i+1}^\top = \big(\mathcal{I} - \alpha \boldsymbol{\phi}_i \boldsymbol{\phi}_i^\top\big) F_i^\top + \alpha \boldsymbol{e}_{i+1} [\gamma_t \boldsymbol{\phi}_{i+1} - \boldsymbol{\phi}_i]^\top + \alpha \boldsymbol{\phi}_i \boldsymbol{\phi}_i^\top , \tag{10}$$

with $\boldsymbol{b}_0 = \boldsymbol{\theta}_{init}$ and $F_0^\top = \mathbf{0}$, and

$$\boldsymbol{e}_{i+1} = \big(\mathcal{I} - \alpha \boldsymbol{\phi}_i \boldsymbol{\phi}_i^\top\big) \gamma_i \lambda \boldsymbol{e}_i + \boldsymbol{\phi}_i ,$$

with $\boldsymbol{e}_0 = \mathbf{0}$. Note that for $\lambda = 0$, $\boldsymbol{e}_{i+1} = \boldsymbol{\phi}_i$ and equations (9) and (10) reduce to equations (5) and (6).

Equation (10) can be simplified by defining $A_i := (\mathcal{I} - F_i^\top)/\alpha$, and rewriting the update for $F^\top$ as an update for A:

$$\begin{aligned}
(\mathcal{I} - F_{i+1}^\top)/\alpha &= \mathcal{I}/\alpha - \big(\mathcal{I} - \alpha \boldsymbol{\phi}_i \boldsymbol{\phi}_i^\top\big) F_i^\top/\alpha \\
&\quad - \boldsymbol{e}_{i+1} [\gamma_{i+1} \boldsymbol{\phi}_{i+1} - \boldsymbol{\phi}_i]^\top - \boldsymbol{\phi}_i \boldsymbol{\phi}_i^\top \\
&= (\mathcal{I} - F_i^\top)/\alpha + \boldsymbol{\phi}_i \boldsymbol{\phi}_i^\top F_i^\top \\
&\quad - \boldsymbol{e}_{i+1} [\gamma_{i+1} \boldsymbol{\phi}_{i+1} - \boldsymbol{\phi}_i]^\top - \boldsymbol{\phi}_i \boldsymbol{\phi}_i^\top \\
&= (\mathcal{I} - F_i^\top)/\alpha - \boldsymbol{\phi}_i \boldsymbol{\phi}_i^\top (\mathcal{I} - F_i^\top) \\
&\quad - \boldsymbol{e}_{i+1} [\gamma_{i+1} \boldsymbol{\phi}_{i+1} - \boldsymbol{\phi}_i]^\top \\
&= \big(\mathcal{I} - \alpha \boldsymbol{\phi}_i \boldsymbol{\phi}_i^\top\big) (\mathcal{I} - F_i^\top)/\alpha \\
&\quad + \boldsymbol{e}_{i+1} [\boldsymbol{\phi}_i - \gamma_{i+1} \boldsymbol{\phi}_{i+1}]^\top \\
A_{i+1} &= \big(\mathcal{I} - \alpha \boldsymbol{\phi}_i \boldsymbol{\phi}_i^\top\big) A_i \\
&\quad + \boldsymbol{e}_{i+1} [\boldsymbol{\phi}_i - \gamma_{i+1} \boldsymbol{\phi}_{i+1}]^\top .
\end{aligned}$$

Because $F_0^\top = \mathbf{0}$, $A_0 = (\mathcal{I} - F_0^\top)/\alpha = \mathcal{I}/\alpha$. In addition, we define $\boldsymbol{d}_i := \boldsymbol{b}_i/\alpha$ to obtain the update:

$$\boldsymbol{d}_{i+1} = \big(\mathcal{I} - \alpha \boldsymbol{\phi}_i \boldsymbol{\phi}_i^\top\big) \boldsymbol{d}_i + \boldsymbol{e}_{i+1} R_{i+1} ,$$

with $\boldsymbol{d}_0 = \boldsymbol{\theta}_{init}/\alpha$. Finally, Equation (4) can be rewritten in terms of A_t and $\boldsymbol{d}_t$:

$$\begin{aligned}
\boldsymbol{\theta}_{(t)} &= \boldsymbol{b}_t + F_t^\top \boldsymbol{\theta}_* \\
&= \alpha \boldsymbol{d}_t + (\mathcal{I} - \alpha A_t) \boldsymbol{\theta}_* \\
&= \boldsymbol{\theta}_* + \alpha (\boldsymbol{d}_t - A_t \boldsymbol{\theta}_*) .
\end{aligned}$$

The method implementing these updates is shown in Algorithm 7. We added two generalizations: we allow the matrix A and vector $\boldsymbol{d}$ to be initialized randomly, and we allow the step-size used in the update of the model to be different than the step-size used in the planning updates. The former, we indicate by β; the latter by α. The reason for making this distinction is that α and β influence

the learning function is very different ways. The parameter β is a forgetting parameter that determines how easily old information is overwritten by new information. It directly influences the model and hence the fixed point. On the other hand, α is a parameter that influences the iterative process for finding this fixed point. Its value determines if, and how quickly, this process converges.

We call this method forgetful LSTD(λ) for obvious reasons: when the forgetting parameter β is set equal to 0, and $A_{init} = \mathbf{0}$ and $\boldsymbol{d}_{init} = \mathbf{0}$, the model $\langle A_t, \boldsymbol{d}_t \rangle$ reduces to the model learned by LSTD(λ) (Boyan, 2002). With these settings, the learning function of forgetful LSTD(λ) cannot be reproduced by the replay method (Algorithm 3). Forgetful LSTD(λ) is a special case of the replay method only if $\beta = \alpha$, $A_{init} = \mathcal{I}/\alpha$ and $\boldsymbol{d}_{init} = \boldsymbol{\theta}_{init}/\alpha$. Note as well that LSTD(λ) traditionally solves the model using the inverse of A_t, whereas Algorithm 7 uses an iterative process. While in principle these techniques can be interchanged, an iterative process offers more flexibility and control over computation time, which is important in an online setting.

Algorithm 7 Forgetful LSTD(λ)

INPUT: $\alpha, \beta, \lambda, k, \boldsymbol{\theta}_{init}, \boldsymbol{d}_{init}, A_{init}$
// *For replay-equivalence, use:*
// $\beta \leftarrow \alpha, \quad A_{init} \leftarrow \mathcal{I}/\alpha, \quad \boldsymbol{d}_{init} \leftarrow \boldsymbol{\theta}_{init}/\alpha$
$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta}_{init}, \quad \boldsymbol{d} \leftarrow \boldsymbol{d}_{init}, \quad A \leftarrow A_{init}$
obtain initial $\boldsymbol{\phi}$
$\boldsymbol{e} \leftarrow \mathbf{0}$
Loop:
  obtain next feature vector $\boldsymbol{\phi}'$,γ and reward R
  $\boldsymbol{e} \leftarrow (\mathcal{I} - \beta\boldsymbol{\phi}\boldsymbol{\phi}^\top)\boldsymbol{e} + \boldsymbol{\phi}$
  $A \leftarrow (\mathcal{I} - \beta\boldsymbol{\phi}\boldsymbol{\phi}^\top)A + \boldsymbol{e}(\boldsymbol{\phi} - \gamma\boldsymbol{\phi}')^\top$
  $\boldsymbol{d} \leftarrow (\mathcal{I} - \beta\boldsymbol{\phi}\boldsymbol{\phi}^\top)\boldsymbol{d} + \boldsymbol{e}R$
  $\boldsymbol{e} \leftarrow \gamma\lambda\boldsymbol{e}$
  Repeat k times:
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha(\boldsymbol{d} - A\boldsymbol{\theta})$
  $\boldsymbol{\phi} \leftarrow \boldsymbol{\phi}'$

5.2. Computational Considerations

First, we consider the time complexity for learning the model. While the update of $\boldsymbol{d}$ in Algorithm 7 is expressed in terms of matrices, it can simply be changed to an update that only involves vectors by using $(\mathcal{I} - \alpha\boldsymbol{\phi}\boldsymbol{\phi}^\top)\boldsymbol{d} = \boldsymbol{d} - \alpha\boldsymbol{\phi}(\boldsymbol{\phi}^\top\boldsymbol{d})$. Hence, the cost of this update is only $\mathcal{O}(n)$, with n the total number of features. A similar argument holds for $\boldsymbol{b}$. The update of A is more expensive. Using sparse feature vectors, its cost is $\mathcal{O}(nm)$, where m is the number of *active features*, that is, the number of non-zero elements of a feature vector.

The planning updates of Algorithm 7 have a time complexity of $\mathcal{O}(n^2k)$ per time step, which is considerably more than the cost for learning the model. However, there are many different iterative techniques that can be used to estimate the solution of A and $\boldsymbol{d}$. For example, at each iteration, only a single feature can be updated, as is done in iLSTD (Geramifard et al., 2006). In this case, the time complexity reduces to $\mathcal{O}(nk)$. Using this strategy while setting k equal to m results in a total time complexity (for learning plus planning) of $\mathcal{O}(nm)$ per time step.

The memory required for storing the model depends on the sparsity of A. In the worst case, the memory requirements are $\mathcal{O}(n^2)$.

5.3. Extension to Control

One of the main advantages of having a prediction method that gracefully forgets old information when new information comes in is that such predictions are well suited for control. In control tasks, forgetting is vital, because the policy changes over time. Hence, old samples are no longer representative for the current policy and can inhibit policy improvement when maintained in the model.

Forgetful LSTD(λ) can be turned into a control method with minimal modification. Simply using feature vectors based on state-action pairs, $\boldsymbol{\phi}_t = \boldsymbol{\phi}(S_t, A_t)$, instead of only states is sufficient; the same update rules can be used. With these features, values can be computed that predict the return for separate actions in a state (i.e., the *Q-values*), which can then be used for any exploration strategy, for example ϵ-greedy or some *softmax* selection strategy. Applying this strategy to forgetful LSTD(λ) results in an on-policy control method that we call *LS-Sarsa(λ)*.

5.4. Mountain Car Experiment

To demonstrate the importance of a multi-step models, we performed a comparison on the mountain car task (Sutton & Barto, 1998), in which an underpowered car has has to move itself out of a valley. [1] Our mountain car task implementation is as described by Sutton and Barto (1998), but we made the domain more challenging by using coarser tile coding: 3 tilings, each consisting of 3-by-3 tiles.

The learning methods used are LS-Sarsa(λ) with $\lambda = 0$ and $\lambda = 0.95$. We used $\alpha = 0.01/3$, $k = 1$ and $\boldsymbol{\theta}_{init} = \mathbf{0}$, and ϵ-greedy exploration with $\epsilon = 0.01$. In addition, we used the settings $\beta = \alpha$, $\boldsymbol{d}_{init} = \boldsymbol{\theta}_{init}/\alpha$ and $A_{init} = \mathcal{I}/\alpha$. At these settings, LS-Sarsa(0) reduces to the on-policy control version of Algorithm 2, based on the linear Dyna model.

In our experiments, we measured the number of steps to the end of an episode for the first 2000 episodes, averaged over 100 independent runs. We set the maximum length of an episode at 10,000 steps. Figure 1 shows the results.

[1] The code for this experiment can be found on https://github.com/vanseijen/singlestep-vs-multistep.

Note that the graph of linear Dyna contains performance jumps with a size of roughly 100 steps, which corresponds with 1 run out of the 100 runs hitting the 10,000 steps limit. This illustrates very clearly the stability issues that one-step models can cause.[2]

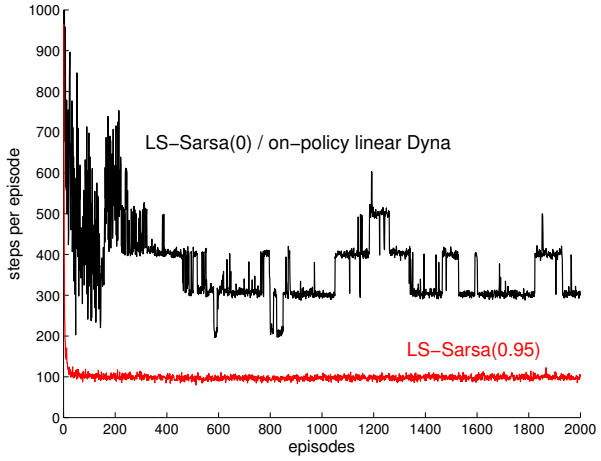

Figure 1. Illustration of the importance of multi-step models. The figure shows the steps per episode on the mountain car control task using three 3-by-3 tilings for LS-Sarsa(λ) with $\lambda = 0$, for which the algorithm reduces to on-policy linear Dyna, and $\lambda = 0.95$. The graphs are averaged over 100 independent runs.

6. Related Work

The replay strategy described in Section 3.1 has been previously mentioned in the context of best-match learning (van Seijen et al., 2011). In best-match learning it is used to combine a partial, tabular model with a model-free, tabular value function in a principled way. We use it here to derive an equivalence relation between model-free updates with linear function approximation and model-based learning based on a linear model. In addition, we show that the replay strategy can be combined with multi-step update targets.

The importance of multi-step models for obtaining robust performance in problems with state aggregation was first observed by Sutton (1995). The idea of a multi-step model was extended to problems that use linear function approximation by Yao et. at. (Yao et al., 2009). However, their multi-step model is constructed by learning a one-step model and iterating this model a number of times. While this approach captures some aspects of multi-step models, it fails to capture the important robustness aspect because at its core it still learns a one-step model.

Least-squares policy iteration (LSPI) (Lagoudakis & Parr, 2003) is another control method related to LSTD. LSPI computes an optimal control policy given a set a samples and an initial policy. It achieves this by repeatedly performing LSTD-like evaluations of its current policy, combined with greedy policy improvements, until the policy has converged. LSPI is an off-policy method based on a one-step model/one-step update targets; by contrast, LS-Sarsa(λ) is an on-policy method based on a multi-step model/multi-step update targets.

[2]We performed additional experiments using other step-sizes, but in none of these experiments did the performance of LS-Sarsa(0) come close to that of LS-Sarsa(λ).

7. Future Work

While we focussed only on linear function approximation, the ideas naturally extend to non-linear function approximation. It simply requires a new implementation of the replay method (Algorithm 4), such that it is based on the update for non-linear function approximation,

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha \left[u_t - \hat{v}(S_t, \boldsymbol{\theta}_t)\right] \nabla_{\boldsymbol{\theta}_t} \hat{v}(S_t, \boldsymbol{\theta}_t) \,,$$

as well as a new implementation of the compute_targets method.

Another interesting research direction is to look at different versions of the update_weights method (Algorithm 6). Different versions of this method correspond with different types of partial models. For example, only specific features could be updated. Using partial models not only reduces the compute time, but also the memory requirements.

8. Conclusion

We showed that there is a close relation between replaying experience and exploiting a model. Specifically, we showed that there are instances where replaying the past results in the same learning function as planning with a learned model. This insight challenges the traditional view on planning and can be used to derive new model-based methods. We demonstrated this by deriving a model-based method, forgetful LSTD(λ), that combines a multi-step model with graceful forgetting. To demonstrate the importance of multi-step models we applied this method to a small control problem with substantial function approximation. Whereas using a multi-step model resulted in fast convergence, the method using a one-step model failed to perform consistently.

9. Acknowledgements

The authors thank Joseph Modayill, Kavosh Asadi, Hado van Hasselt and Rupam Mahmood for insights and discussions contributing to the results presented in this paper. We also gratefully acknowledge funding from Alberta Innovates – Technology Futures and from the Natural Sciences and Engineering Research Council of Canada.

References

Adam, S., Busoniu, L., and Babuska, R. Experience replay for real-time reinforcement learning control. *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews*, 42(2):201–212, 2012.

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

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

Geramifard, A., Bowling, M., and Sutton, R. S. Incremental least-squares temporal difference learning. In *Proceedings of the 21st national conference on Artificial intelligence*, pp. 356–361, 2006.

Kaelbling, L. P., Littman, M. L., and Moore, A. P. Reinforcement learning: a survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.

Lagoudakis, M. G. and Parr, R. Least-squares policy iteration. *Journal of Machine Learning Research*, 4:1107–1149, 2003.

Lin, L. J. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine Learning*, 8:293–321, 1992.

Parr, R., Li, L., Taylor, G., Painter-Wakefield, C., and Littman, M.L. An analysis of linear models, linear value-function approximation, and feature selection for reinforcement learning. In *Proceedings of the 25th international conference on Machine learning (ICML)*, pp. 752–759, 2008.

Riedmiller, M. Neural fitted Q iteration — first experiences with a data efficient neural reinforcement learning method. In *Proceedings of the 16th European conference on machine Learning*, 2005.

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

Sutton, R. S. TD models: Modeling the world at a mixture of time scales. In *Proceedings of the 12th International Conference on Machine Learning (ICML)*, pp. 531–539, 1995.

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

Sutton, R. S., Szepesvári, C., Geramifard, A., and Bowling, M. Dyna-style planning with linear function approximation and prioritized sweeping. In *International Conference on Uncertainty in Artificial Intelligence (UAI)*, pp. 528–536, 2008.

Szepesvári, C. *Algorithms for reinforcement learning*. Morgan and Claypool, 2009.

van Seijen, H. H. and Sutton, R. S. True online TD(λ). In *Proceedings of the 31th international conference on Machine learning (ICML)*, 2014.

van Seijen, H. H., Whiteson, S., van Hasselt, H., and Wiering, M. Exploiting Best-Match Equations for Efficient Reinforcement Learning. In *Journal of Machine Learning Research*, 12:2045–2094, 2011.

Yao, H., Bhatnagar, S., and Diao, D. Multi-step linear dyna-style planning. In *Advances in Neural Information Processing Systems 22 (NIPS)*, pp. 2187–2195, 2009.