

# STD( $\lambda$ ): learning state differences with TD( $\lambda$ )

**Lex Weaver**

Department of Computer Science  
Australian National University  
ACT AUSTRALIA 0200  
*Lex.Weaver@anu.edu.au*

**Jonathan Baxter**

WhizBang! Labs  
4616 Henry Street  
Pittsburgh PA USA 15213  
*JBaxter@whizbang.com*

## Abstract

TD( $\lambda$ ) with function approximation has proved empirically successful for some complex reinforcement learning problems. For linear approximation, TD( $\lambda$ ) has been shown to minimise the squared error between the approximate value of each state and the true value. However, as far as policy is concerned, it is error in the relative ordering of states that is important, rather than error in the state values. We illustrate this point with a simple two-state system in which TD( $\lambda$ ) abandons the optimal policy to converge to a suboptimal policy. We also observe this trait of policy degradation by TD( $\lambda$ ) in backgammon. We then present a modified form of TD( $\lambda$ ), called STD( $\lambda$ ), in which function approximators are trained with respect to relative state values. We characterise the limiting behaviour of this algorithm, and present a theorem guaranteeing optimality of the limiting parameter vector for two-state BMDPs. A comparison with Bertsekas' *differential training* method is also presented, which highlights a significant difference between the algorithms. This is followed by successful demonstrations of STD( $\lambda$ ) on the two-state system and the well known acrobot problem.

## 1 Introduction

For complex reinforcement learning problems, TD( $\lambda$ ) with function approximation [Sutton, 1988] has proved empirically successful. Its origins go back as far as Samuel's Checkers Program [Samuel, 1959], while perhaps its most famous success has been Tesauro's TD-Gammon [Tesauro, 1992; 1994]. A variant of TD( $\lambda$ ) for minimax search has also been successful in learning to play chess [Baxter *et al.*, 2000].

For linear approximation, TD( $\lambda$ ) has been shown to minimise the squared error between the approximate value of each state and the true value [Tsitsikilis and Van Roy, 1997; Dayan, 1992; Gordon, 1995; Singh *et al.*, 1995]. However, as far as policy is concerned, it is error in the relative ordering of states that is critical, rather than error in the state values. Consider a simple system consisting of two-states:  $A$  and  $B$ , where the value of each state under the optimal policy is 10 and 5 respectively. A function approximator which estimated

the state values to be 15 and 0, would implement the optimal policy whilst having a squared error of  $5^2$ . However, a function approximator estimating the values as 7 and 8, would have a squared error of only  $3^2$ , yet would not implement the optimal policy.

We illustrate this point further in section 2 with a simple two-state system in which TD( $\lambda$ ), starting from the optimal policy, converges to a suboptimal policy. In section 3 we demonstrate that this also occurs in a more complex system: backgammon. In section 4 we present a modified form of TD( $\lambda$ ), called State-Temporal Difference( $\lambda$ ) (or simply STD( $\lambda$ )), in which function approximators are trained with respect to relative state values on binary decision problems. We characterise the limiting behaviour of STD( $\lambda$ ) and provide a theorem guaranteeing optimality of the limiting parameter vector for two-state BMDPs. Section 4.2 compares STD( $\lambda$ ) with *differential training* [Bertsekas, 1997], highlighting the different distributions of state pairs used in training. Experimental results in section 5 demonstrate the success of STD( $\lambda$ ) in the two-state system and in a variant of the well known acrobot problem.

### 1.1 Previous work on State Differences.

Several researchers have previously considered using state difference information. Utgoff and Saxena [Utgoff and Saxena, 1987] described the Best-First Preference Learning Algorithm for creating a set of preference predicates from a previously known solution. Utgoff and Clouse [Utgoff and Clouse, 1991] showed how a similar algorithm can be used to generate weight updates for a linear function approximator. The technique relies upon an external expert to nominate the preferred states, and thus is a form of supervised learning. They have also constructed an algorithm which has a supervised learning phase to utilise expert preferences, and a reinforcement learning phase to make traditional TD-style updates which don't use the expert preferences. Tesauro [Tesauro, 1989] described an experiment in which he used expert preferences to train a neural network to choose between backgammon positions.

The approximation of cost-to-go differences—rather than just the cost-to-go—has been investigated by a number of authors in the context of  $Q$ -learning. Werbos [Werbos, 1990; 1992] discusses the merits of approximating the gradient of the  $Q$  function, whilst Baird [Baird, 1993], Harmon, and

Current State	Action	Destination State Probabilities	
		A	B
A or B	$a_1$	$\frac{1}{3}$	$\frac{2}{3}$
A or B	$a_2$	$\frac{2}{3}$	$\frac{1}{3}$

Table 1: Transition Matrix of the two-state System

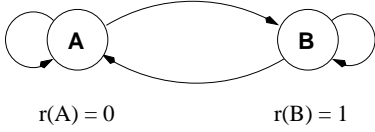


Figure 1: Transitions and rewards in the two-state system.

Klopf [Harmon *et al.*, 1994] introduced *advantaging updating* which estimates the value of each state and the relative advantage of each action using separate approximation architectures.

More recently McGovern and Moss [McGovern and Moss, 1998] have used temporal difference learning to develop an instruction scheduler for an optimising compiler. Their approach uses table-lookup rather than function approximation, and combines possible successor states into a single feature vector which is mapped to a preference indicator.

Bertsekas' *differential training* [Bertsekas, 1997] is the most closely related previous work. We defer discussion of it until section 4.2.

## 2 Generating Suboptimal Policies

Convergence by TD( $\lambda$ ) to suboptimal policies can be found in even the simplest non-trivial system — a Markov Decision Process with only two states. Consider the transition matrix shown in Table 1. The corresponding machine, shown in Figure 1, has two states, A and B, and in each state there are two actions to choose from. Action  $a_1$  takes the machine to state B with probability  $\frac{2}{3}$  and action  $a_2$  takes it to state A with the same probability.

We denote the reward for entering state  $x$  by  $r(x)$ , and define  $r(A) = 0$ , and  $r(B) = 1$ . If we assume only deterministic policies, there are four possible policies for this system: always choose action  $a_1$ , always choose action  $a_2$ , choose  $a_1$  in state A and  $a_2$  in state B, or vice-versa.

In this paper we only consider infinite-horizon, discounted-reward problems, so the value of a state  $x$  under policy  $\mu$ ,  $J^\mu(x)$ , is defined to be:

$$J^\mu(x) = \mathbb{E}^\mu \left[ \sum_{t=0}^{\infty} \alpha^t r(x_t) \mid x_0 = x \right],$$

where  $x_t$  is the current state at time  $t$ ,  $\mathbb{E}^\mu$  denotes the expectation over trajectories  $x_0, x_1, \dots$  under policy  $\mu$ , and  $\alpha \in [0, 1)$  is the discount factor.

For this simple system the optimal policy is to always choose the action leading to state B with highest probability, that is to choose action  $a_1$ . Let  $J^*(x)$  denote the value of state  $x$  under the optimal policy. The following relations between  $J^*(A)$  and  $J^*(B)$  must hold:

$$J^*(A) = r(A) + p(A, A)\alpha J^*(A) + p(A, B)\alpha J^*(B)$$

$$J^*(B) = r(B) + p(B, A)\alpha J^*(A) + p(B, B)\alpha J^*(B),$$

where  $p(x, x')$  denotes the probability of a transition from state  $x$  to  $x'$  under the optimal policy.

Substituting for  $p(\cdot)$  from Table 1 and simplifying, we can express  $J^*(\cdot)$  in terms of  $\alpha$ :

$$J^*(A) = \frac{2\alpha}{3(1-\alpha)} \quad (1)$$

$$J^*(B) = 1 + \frac{2\alpha}{3(1-\alpha)} \quad (2)$$

Note that, as expected,  $J^*(B) > J^*(A)$  for all  $\alpha \in [0, 1)$ .

We define our approximate linear value function  $\tilde{J}(x, w)$  by

$$\tilde{J}(x, w) = w \cdot \phi(x) = \sum_{i=1}^k w_i \phi_i(x),$$

where  $w = (w_1, \dots, w_k)$  is the parameter vector, and  $\phi(x) = (\phi_1(x), \dots, \phi_k(x))$  is a function mapping states to feature vectors. In our simple two state system we require the dimensionality of  $w$  and  $\phi$  to be 1 (*i.e.*  $k = 1$ ) in order to ensure that  $\tilde{J}$  cannot approximate all possible value functions (*i.e.* to ensure we really are in an *approximate* value-function setting). Hence our approximate value function is simply  $\tilde{J}(x, w) = w\phi(x)$  for scalar  $w$  and  $\phi$ . For the two state example we take  $\phi(A) = 2$  and  $\phi(B) = 1$ .

If we generate a policy from  $\tilde{J}(x, w)$  by using one-step greedy look-ahead, then to prefer action  $a_1$  over action  $a_2$ , we require  $\tilde{J}(B, w) > \tilde{J}(A, w)$ , which will hold only if  $w < 0$ . Hence, for a learning algorithm to yield an approximator which implements the optimal policy (*i.e.* correctly values state B above state A), it must tune  $w$  to a negative value.

Assuming the optimal policy is being followed, and with TD(1) as the learning algorithm, [Tsitsiklis and Van Roy, 1997, Theorem 1] shows that  $w$  will converge to

$$w_\infty = \operatorname{argmin}_w \|\tilde{J}(\cdot, w) - J^*\|_D,$$

where

$$\|\tilde{J}(\cdot, w) - J^*\|_D = E_{TD}(w),$$

with  $E_{TD}(w)$  — the error function — defined as

$$E_{TD}(w) = \sum_x \pi(x) (\tilde{J}(x, w) - J^*(x))^2, \quad (3)$$

in which  $\pi(x)$  is the stationary probability of state  $x$ . That is, TD(1) converges to a parameter vector minimising the weighted least-squared error between the approximate value of a state and its true value, where each state is weighted by its stationary probability.

With the stationary distribution,  $[\pi(A), \pi(B)] = [\frac{1}{3}, \frac{2}{3}]$ , we can find the  $w_\infty$  minimising (3) by solving  $\frac{dE_{TD}(w)}{dw} = 0$  for  $w$ , where

$$\begin{aligned} \frac{dE_{TD}(w)}{dw} &= 2\pi(A)[w\phi(A) - J^*(A)]\phi(A) \\ &\quad + 2\pi(B)[w\phi(B) - J^*(B)]\phi(B). \end{aligned}$$

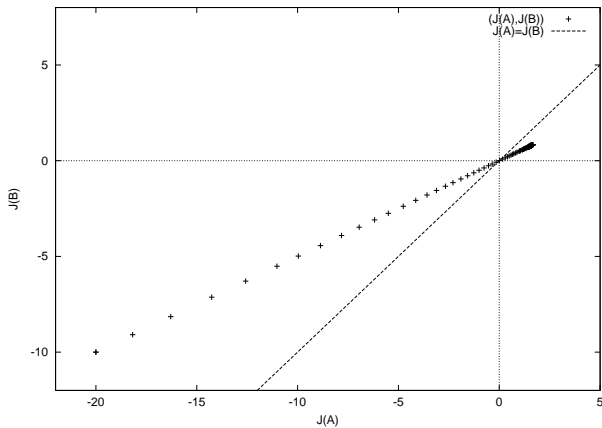


Figure 2: Evolution of  $(\tilde{J}(A, w), \tilde{J}(B, w))$  using TD(1) and starting from  $w = -10$ , *i.e.*  $(-20, -10)$ , plotted every 50 iterations. The region above  $J(A) = J(B)$  gives optimal policy.

For  $\alpha = \frac{1}{2}$ , a quick calculation gives  $w_\infty = \frac{7}{9}$  — the wrong sign. Figure 2 shows the evolution of  $(\tilde{J}(A, w), \tilde{J}(B, w))$  plotted after every 50 iterations of TD(1). The system converges to  $(\frac{14}{9}, \frac{7}{9})$ , minimising the squared error to  $J^*(A) = \frac{6}{9}$  and  $J^*(B) = \frac{15}{9}$  — the true values under the optimal policy.

Note that the system starts in the region of optimal policy, with  $w < 0$  and  $(\tilde{J}(A, w), \tilde{J}(B, w))$  above the line  $J(A) = J(B)$ . However, TD(1) moves it into the suboptimal policy region to minimise the squared error on the state value approximations.

The reason for this is perhaps best understood geometrically. Figure 3 shows the space of all value functions for our system, with axes weighted by the stationary distribution of the corresponding state under the optimal policy, *i.e.* the  $x$ -coordinate is weighted by  $\frac{1}{3}$  and the  $y$ -coordinate by  $\frac{2}{3}$ . Value functions in the lighter region value state B above state A, and hence generate an optimal greedy policy, while those in the darker region generate suboptimal policies. The boundary crossing line represents the set of realisable approximate value functions  $(w\phi(A), w\phi(B))$ .

The TD(1) convergence point is then simply the approximate value function found by projecting the true value function  $(J^*(A), J^*(B))$  onto the set of realisable functions. As can be seen in Figure 3, TD(1) minimises the weighted distance between the true value function and the convergence point, but takes no account of the of the policy boundary and ends up in the suboptimal policy region.

### 3 Suboptimal Policies in Backgammon

The example of the previous section demonstrates that it is possible for TD( $\lambda$ ) to degrade the policy whilst still minimising  $\|\tilde{J} - J^\mu\|_D$ , where  $J^\mu$  is the true value function of the system being observed.<sup>1</sup> We now show that this behaviour

<sup>1</sup>If the optimal policy is being observed, as is the case in the previous section, then  $J^\mu = J^*$ .

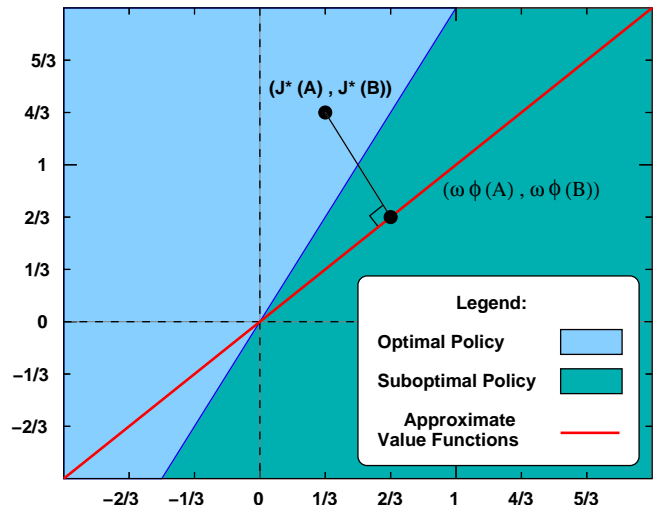


Figure 3: A geometric interpretation of TD(1)'s behaviour in the two-state system.

affects not only artificial examples, but is also evident in a real domain: backgammon — where TD( $\lambda$ ) has had its most famous success.

Our backgammon playing program has been created along the lines of Tesauro's TD-Gammon [Tesauro, 1992; 1994]. Its neural network function approximator has 209 input nodes, no hidden layer, and 1 output node, which is a squashed linear function of the inputs. The input vector consists of 200 elements directly representing the board, 8 elements representing hand-coded features extracted from the board (*e.g.* probability of hitting a blot), and a constant value 1.

We used TD( $\lambda$ ) to train a large number of randomly generated neural networks using self-play, and monitored how their level of performance changed as training progressed. We measured performance by playing the network against a hand-coded fixed opponent for at least 2000 games, recording the proportion of games won, ignoring gammons and backgammons. This fixed opponent was based on Tesauro's PUBEVAL.<sup>2</sup> About half of the runs exhibited an interesting behaviour, whereby performance initially increases, but then suffers a noticeable drop before recovering to plateau at about 0.8 (winning 4 games in 5) against the benchmark opponent. The other runs show generally monotonic performance increases, but also plateau at about 0.8.

To ensure that this behaviour was not an artefact of the benchmark opponent, we re-generated the performance curves with other benchmark opponents. These were created by randomly generating function approximators and training them until they were of similar strength<sup>3</sup> to the fixed opponent. The results were unchanged, the individual curves shifting up or down slightly with the variation in the playing strength of the benchmark opponent, but the characteristic

<sup>2</sup>Available from <http://www-cgi.cs.cmu.edu/afs/cs.cmu.edu/project/connect/code/tesauro>

<sup>3</sup>This is necessary because if they were significantly weaker or stronger, the detail of the effect we wish to observe would be obscured as the curve is compressed towards the top or bottom axis.

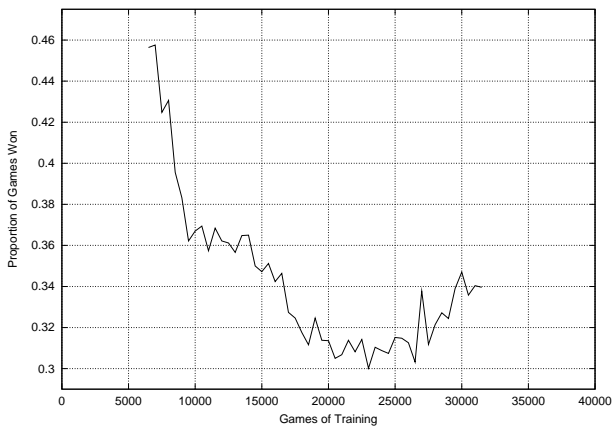


Figure 4: Performance of function approximator for backgammon, as it trains by observation with TD( $\lambda$ ). Learning rate = 0.003,  $\lambda = 0.0$

policy degradation remained.

To be certain that it is the effect observed in the previous section, we need to show that TD( $\lambda$ ) is making progress in minimising  $\|\tilde{J} - J^\mu\|_D$  whilst the policy is degrading. This is not as straightforward in the case of a backgammon system using TD( $\lambda$ ) to train by self-play, because TD( $\lambda$ ) does not passively observe the play, but rather regularly modifies the player. Hence the policy  $\mu$  changes each time the neural network weights are updated, and so the existing theoretical results no longer apply. To overcome this, we modified our system to make the TD( $\lambda$ ) updates on the parameters of a non-playing second approximator which was initialised as a copy of the one being observed.

We used this modified system to train a network taken from the original experiment. The network selected had originally been trained for 6500 games (the performance peak prior to degradation setting in) and was chosen because it was one of the group of networks which went on to exhibit policy degradation, *i.e.* training it under the original TD( $\lambda$ ) regime had resulted in networks implementing inferior policies. We called this network  $w_{start}$ .

To estimate  $\|\tilde{J} - J^\mu\|_D$  we need to estimate both  $J^\mu$  and the distribution  $D$ . Since the performance of the network is being compared to the fixed opponent,  $D$  is the distribution over states generated by play between  $w_{start}$  and this opponent. So to estimate  $D$ , we collected a large number (2000) positions  $S := \{x_1, \dots, x_n\}$  from many games between  $w_{start}$  and the fixed opponent. Each position  $x \in S$  was then rolled-out 200 times, and the results averaged to form an estimate  $\hat{J}(x)$  of  $J^\mu(x)$ . Thus  $\|\tilde{J} - J^\mu\|_D$  is estimated by  $\|\tilde{J} - \hat{J}\|_{\hat{D}}$ , where

$$\|\tilde{J} - \hat{J}\|_{\hat{D}} := \frac{1}{n} \sum_{x \in S} [\tilde{J}(x, w) - \hat{J}(x)]^2.$$

Note that  $\|\tilde{J} - \hat{J}\|_{\hat{D}}$  is an unbiased estimate of  $\|\tilde{J} - J^\mu\|_D$ .

Figure 4 shows a performance curve for  $w_{start}$  against the fixed opponent. Performance drops from a high of 0.46 to a

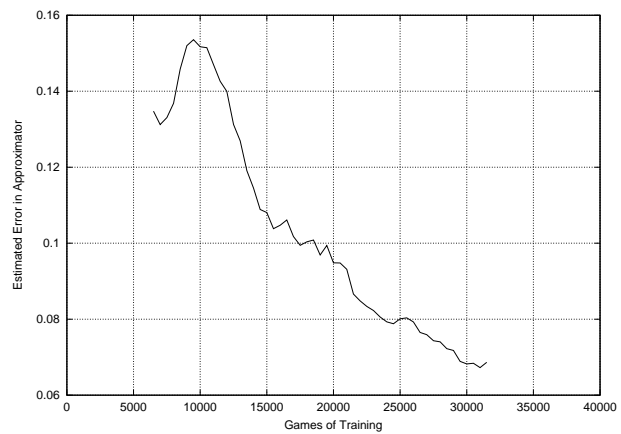


Figure 5: Estimated error in the function approximator for backgammon, as it trains by observation with TD( $\lambda$ ). Learning rate = 0.003,  $\lambda = 0.0$ .

low of 0.30, a decline of about one-third.

In Figure 5 we see that our estimate of  $\|\tilde{J} - J^\mu\|_D$  (called Estimated Error in Approximator), rises initially and then declines throughout the remainder of the run.

Together these curves are extremely interesting. Normally we would expect to find the error falling with the policy improving concurrently. However, the figures show not this, but two different types of behaviour. Firstly, we see the error rising whilst policy degrades, and secondly we see error falling rapidly as policy continues to degrade.

The first behaviour is not unexpected. If we believe that policy improves as error declines, we should also believe that policy will degrade as error increases. The second behaviour however, clearly demonstrates the policy degradation effect — approximation error falls by half, whilst the performance of the policy continues to decline. Hence, in this case, minimising error in the state-value approximations leads to an inferior policy.

## 4 Improving Policy - STD( $\lambda$ )

The results of the previous two sections raise the question of whether there is a better way of doing TD-style updates. Is there an update rule which precludes policy degradation?

The problem with TD( $\lambda$ ) is that it takes no account of the policy implemented by the function approximator. The updates are not directly derived from the policy, and there is no consideration of the effect on policy that the updates will have. The updates are calculated with the sole intention of moving  $\tilde{J}$  closer to  $J^\mu$ , but as we have seen in section 2, the region around  $J^\mu$  may implement suboptimal policies. Even if the region to which we converge has good policies, as in the case of backgammon (section 3), the path which TD( $\lambda$ ) follows may traverse regions of bad policy.

The idea we have tried to capture with our algorithm, STD( $\lambda$ ), is to use the temporal differences to learn relative state values. This endeavours to improve the policy directly, rather than concentrating on approximating individual state values. To simplify matters we have restricted our attention

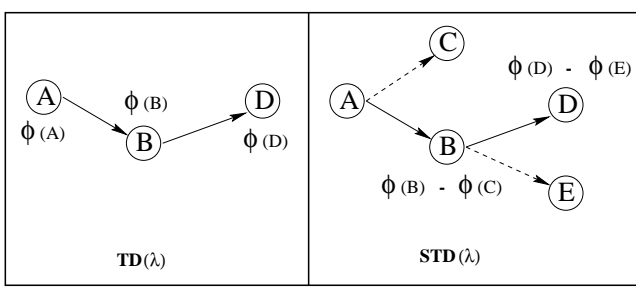


Figure 6: A sequence of states in a Binary Markov Decision Process (BMDP) and how they are viewed by  $\text{TD}(\lambda)$  and  $\text{STD}(\lambda)$ . The solid lines represents the path taken by the process,  $A \rightarrow B \rightarrow D$ , with the dotted line representing the alternative paths that might have been followed.  $\text{TD}(\lambda)$  only sees the states the system actually visited ( $A, B, D$ ), and makes updates based on the feature vectors of those states ( $\phi(A), \phi(B), \phi(D)$ ).  $\text{STD}(\lambda)$  knows about the alternative paths, or *sibling states*, and makes updates based on the *differences* between feature vectors of sibling states ( $\phi(B) - \phi(C), \phi(D) - \phi(E)$ ).

to *Binary Markov Decision Processes* (BMDPs) — loosely speaking, at every state in a BMDP there are at most two states that the system can go to next. We call these states *sibling states*. Unlike  $\text{TD}(\lambda)$  which operates on the feature vectors of states directly,  $\text{STD}(\lambda)$  operates on the *difference* between feature vectors of sibling states. This is illustrated in Figure 6.

#### 4.1 The $\text{STD}(\lambda)$ algorithm

We consider a linear approximation to  $J^*$  of the form

$$\tilde{J}(i, w) := \sum_{d=1}^k w_d \phi_d(i), \quad (4)$$

where  $w := (w_1, \dots, w_k)$  is a vector of tunable parameters and  $\phi(i) := (\phi_1(i), \dots, \phi_k(i))$  is the feature vector associated with state  $i$ .

The  $\text{STD}(\lambda)$  algorithm is described in Algorithm 1. Note that we can obtain a version of  $\text{STD}(\lambda)$  for nonlinear  $\tilde{J}(i, w)$  by replacing step 6 with

$$z_{t+1} := \alpha \lambda z_t + \frac{\nabla \tilde{J}(x_{t+1}, w) - \nabla \tilde{J}(x'_{t+1}, w)}{p^\mu(x_t, x_{t+1})} \quad (5)$$

We now make the following assumptions:

**Assumption 1.** The step sizes  $\gamma_t$  are positive and predetermined with  $\sum_{t=0}^{\infty} \gamma_t = \infty$  and  $\sum_{t=0}^{\infty} \gamma_t^2 < \infty$ .

**Assumption 2.** The Markov chain generated by the BMDP under policy  $\mu$  has a unique stationary distribution  $\pi = (\pi(1), \dots, \pi(n))$ .

**Assumption 3.** The matrix

$$\Phi := \begin{pmatrix} \phi_1(1) & \cdots & \phi_d(1) \\ \vdots & \ddots & \vdots \\ \phi_1(n) & \cdots & \phi_d(n) \end{pmatrix}$$

has full rank.

#### Algorithm 1 The $\text{STD}(\lambda)$ algorithm

1: **Given:**

- $\lambda \in [0, 1], \alpha \in [0, 1)$
- State sequence  $x_0, x_1, \dots$  generated by a BMDP under some policy  $\mu$ .
- Step sizes  $\gamma_t > 0$ .
- Linear function approximator  $\tilde{J}(\cdot, w)$  parameterised by  $w \in \mathbb{R}^k$ .

2: Choose any starting state  $x_0$ , initial parameter vector  $w_0$ , and set  $z_0 = 0, (z_0 \in \mathbb{R}^K)$ .

3: **for** each state transition  $x_t \rightarrow x_{t+1}$  **do**

$$4: \quad d_t := [r(x_t) - r(x'_t)] + \alpha \left[ \tilde{J}(x_{t+1}, w_t) - \tilde{J}(x'_{t+1}, w_t) \right] - \left[ \tilde{J}(x_t, w_t) - \tilde{J}(x'_t, w_t) \right]$$

$$5: \quad w_{t+1} := w_t + \gamma_t d_t z_t$$

$$6: \quad z_{t+1} := \alpha \lambda z_t + \frac{\phi(x_{t+1}) - \phi(x'_{t+1})}{p^\mu(x_t, x_{t+1})}$$

$$7: \quad t := t + 1$$

8: **end for**

Since we have assumed the existence and uniqueness of a stationary distribution  $\pi(i)$  over states, there exists a stationary distribution  $\pi(i, i')$  over states and their siblings. That is,  $\pi(i, i')$  is the stationary probability of being in state  $i$ , with  $i'$  as the sibling state. Note that under policy  $\mu$ ,

$$\pi(i, i') = \sum_j \pi(j) p^\mu(j, i)$$

where the sum is over all states  $j$  with successor states  $i$  and  $i'$ , and  $p^\mu(j, i)$  is the probability of making a transition from state  $j$  to state  $i$  under  $\mu$ .

Under the above assumptions, with a linear value function (4), and with a fixed policy  $\mu$ , the sequence of pairs of sibling states  $(x_t, x'_t), (x_{t+1}, x'_{t+1}), \dots$  forms a Markov chain with transition probabilities

$$p^\mu((i, i'), (j, j')) = \begin{cases} p^\mu(i, j) & \text{if both } j \text{ and } j' \text{ can follow } i, \\ 0 & \text{otherwise} \end{cases}$$

and a stationary distribution  $\pi(i, j)$ .

Applying the arguments used to support the convergence of  $\text{TD}(\lambda)$  in the linear case [Tsitsiklis and Van Roy, 1997, Theorem 1], we can see that for  $\text{STD}(\lambda)$ , the limiting vector  $w_\infty$  satisfies

$$\begin{aligned} & \sum_{i,j} \pi(i, j) \left[ (\tilde{J}(i, w_\infty) - \tilde{J}(j, w_\infty)) - (J^\mu(i) - J^\mu(j)) \right]^2 \\ & \leq \frac{1 - \alpha \lambda}{1 - \alpha} \inf_w \sum_{i,j} \pi(i, j) \\ & \quad \left[ (\tilde{J}(i, w) - \tilde{J}(j, w)) - (J^\mu(i) - J^\mu(j)) \right]^2 \end{aligned}$$

Thus, within a factor of  $\frac{1 - \alpha \lambda}{1 - \alpha}$ ,  $\text{STD}(\lambda)$  is aiming for a

parameter vector  $w$  minimising the error function:

$$E_{STD}(w) := \sum_{i,j} \pi(i, j) \left[ (\tilde{J}(i, w) - \tilde{J}(j, w)) - (J^\mu(i) - J^\mu(j)) \right]^2 \quad (6)$$

Compare this with the corresponding error function for TD( $\lambda$ ), equation (3). It is obvious that when  $E_{TD} = 0$  (*i.e.* the function approximator yields the true state values) an optimal greedy policy will be implemented. Similarly, when  $E_{STD} = 0$  (*i.e.* the function approximator yields correct state value differences) an optimal greedy policy will also be implemented.

**Definition 1.** Given two error functions  $E_1$  and  $E_2$ , we say that  $E_1 > E_2$  if  $E_2 = 0 \Rightarrow E_1 = 0$  and  $E_1 = 0 \not\Rightarrow E_2 = 0$ .

True state values necessarily imply correct state value differences, so we have  $E_{TD} = 0 \Rightarrow E_{STD} = 0$ . However, the converse does not hold, as the function approximator may yield correct state value differences without the state values themselves being correct — hence we have  $E_{STD} = 0 \not\Rightarrow E_{TD} = 0$ . For example, in the context of the two-state system of section 2, a function approximator which values state A at -1 and state B at 0, would satisfy  $E_{STD} = 0$  (and implement an optimal policy) without satisfying  $E_{TD} = 0$ . Thus, the error function of STD( $\lambda$ ) admits, as a convergence point of the STD( $\lambda$ ) algorithm, parameterisations of the state value approximator which correctly represent the state differences. This is less strict than the TD( $\lambda$ ) error function which only admits parameterisations which correctly value the states themselves. Hence, we say  $E_{STD} > E_{TD}$ .

The potential significance of  $E_{STD} > E_{TD}$  can be seen in the two state system. TD( $\lambda$ ) endeavours to minimise  $E_{TD}$ , effectively seeking a single  $w$  to solve two equations simultaneously (*i.e.* the values of the two states). In contrast, STD( $\lambda$ ) minimises  $E_{STD}$ , which reduces to seeking a  $w$  to solve the equation

$$\tilde{J}(A) - \tilde{J}(B) = J^\mu(A) - J^\mu(B)$$

which is equivalent to

$$w = \frac{J^\mu(A) - J^\mu(B)}{(\phi(A) - \phi(B))}$$

Note that  $w$  does not depend on the stationary distribution over states. This leads us to the following theorem.

**Theorem 1.** Let there be a two-state BMDP driven by an arbitrary but stationary policy. Let STD(1) observe this process and accordingly tune the parameters of a linear state value function approximator,  $\tilde{J}(\cdot, w_t)$ . Then the limiting parameter vector,  $w_\infty$ , will implement an optimal one-step greedy lookahead policy for the two-state BMDP.

Obviously the same cannot be said of TD(1) (recall the discussion in section 2).

## 4.2 Comparison with Differential Training

As mentioned earlier, the method of *differential training* described by Bertsekas [Bertsekas, 1997] is the most closely related to the STD( $\lambda$ ) algorithm presented here. During training, it requires two instances of the system being observed to

run in lock-step but evolve independently. At time  $t$ , the state of one of the instances (chosen arbitrarily) is referred to as  $x_t$ , with the state of the other instance being  $\hat{x}_t$ . The method uses a TD( $\lambda$ ) approach to learn an approximation,  $\tilde{G}(x, \hat{x}, w)$ , to the true differences in the values of these states:

$$G^\mu(x_t, \hat{x}_t) = J^\mu(x_t) - J^\mu(\hat{x}_t).$$

$G^\mu$  is viewed as the reward function of a problem involving the compound states  $(x, \hat{x})$ , and the reward per stage:  $r(x_t) - r(\hat{x}_t)$ . Hence  $G^\mu$  satisfies the Bellman equation:

$$G^\mu(x_t, \hat{x}_t) = \mathbb{E}\{r(x_t) - r(\hat{x}_t) + \alpha G^\mu(x_{t+1}, \hat{x}_{t+1})\}.$$

We will refer to the differential training algorithm that uses TD( $\lambda$ ) updates in an infinite horizon discounted reward setting as DT( $\lambda$ ).

Assuming  $\tilde{G}$  is linear, and under Assumptions 1–3, DT( $\lambda$ ) will converge (within a factor of  $\frac{1-\alpha\lambda}{1-\alpha}$ ) to a  $w$  which minimises:

$$\begin{aligned} E_{DT}(w) &= \sum_{x, \hat{x}} \pi(x)\pi(\hat{x}) \left[ \tilde{G}(x, \hat{x}, w) - G^\mu(x, \hat{x}) \right]^2 \\ &= \sum_{x, \hat{x}} \pi(x)\pi(\hat{x}) \left[ (\tilde{J}(x, w) - \tilde{J}(\hat{x}, w)) - (J^\mu(x) - J^\mu(\hat{x})) \right]^2 \end{aligned} \quad (7)$$

(by [Tsitsikilis and Van Roy, 1997, Theorem 1]).

Comparing (7) with (6), we see that DT( $\lambda$ ) is minimising the same quantity we minimise with STD( $\lambda$ ), except that DT( $\lambda$ ) is working with a different distribution over state pairs. In particular, the distribution observed by DT( $\lambda$ ) gives a non-zero probability to a pair  $(y, \hat{y})$ , even if the two component states can never occur as siblings. If  $y$  and  $\hat{y}$  are both high probability states, then  $\pi(y)\pi(\hat{y})$  will be relatively large and the evolution of  $w$  will be influenced towards achieving a correct approximation for a pair that will never occur outside the training environment.

In contrast, STD( $\lambda$ ) observes only a single instance of the system during training, and thus trains with respect to the distribution of state pairs occurring under policy  $\mu$ . State pairs whose components can never occur as siblings will not be observed, and will not influence the evolution of  $w$ .

Note that for many dynamical systems (such as physical systems) the number of sibling state-pairs with non-zero probability is  $O(n)$ , while the total number of state-pairs (and hence the number with non-zero probability under DT( $\lambda$ )) is always  $O(n^2)$ . In these systems nearly all the state-pairs considered in DT( $\lambda$ ) are misleading.

## 4.3 Policy degradation in DT( $\lambda$ )

In this section we present an example 3-state system in which DT( $\lambda$ ) converges to a suboptimal policy while STD( $\lambda$ ) converges to the optimal policy. This example is designed simply to illustrate the pitfalls associated with training against the wrong distribution. In particular, we do not claim STD( $\lambda$ ) always dominates DT( $\lambda$ ).

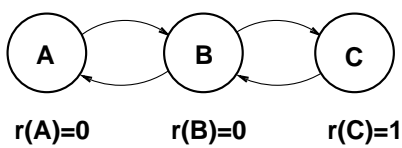


Figure 7: Transitions and rewards in the example three-state system.

The Markov chain with rewards is shown in Figure 7. The only state in which a decision must be made is state  $B$ , and in that state there are two actions:  $a_1$  goes to state  $C$  with probability 0.9 and  $A$  with probability 0.1, while action  $a_2$  goes to state  $C$  with probability 0.1 and state  $A$  with probability 0.9. Clearly the optimal policy is to always choose action  $a_1$ , which gives a stationary distribution of  $\pi(A) = 0.05, \pi(B) = 0.5, \pi(C) = 0.45$ . Under this policy, and for a given discount factor  $\alpha \in [0, 1)$ , the relationships existing between the state values yield the following system of equations.

$$\begin{aligned} J(A) &= \frac{0.9\alpha^2}{1 - \alpha^2} \\ J(B) &= \frac{0.9\alpha}{1 - \alpha^2} \\ J(C) &= 1 + \frac{0.9\alpha^2}{1 - \alpha^2} \end{aligned}$$

If we take our function approximator to be one-dimensional so that  $J(x, w) = w\phi(x)$ , and set  $\phi(A) = 1, \phi(B) = 3$  and  $\phi(C) = 2$ , then any positive weight will value state  $C$  above state  $A$  and hence will implement the optimal policy, while any negative weight will implement the suboptimal policy. A quick calculation shows that the limiting weight  $w_\infty^{\text{DT}}$  of the DT(1) algorithm satisfies

$$w_\infty^{\text{DT}} = \frac{0.18\alpha - 0.81}{1.39(1 + \alpha)}$$

which is negative for all  $\alpha \in [0, 1)$ , *i.e.* the wrong sign, while the limiting weight for STD(1) is  $w_\infty^{\text{STD}} = 1$ , which is the right sign for  $\alpha \in [0, 1)$ , and represents an exact modeling of  $J(C) - J(A)$  by  $\tilde{J}(C) - \tilde{J}(A)$ .

## 5 Experimental Results

The aim of our first set of experiments was simply to illustrate Theorem 1 of section 4.1, *i.e.* when observing a two-state system, STD(1) will always converge to an optimal policy. Using STD(1), the system was started with  $w = 0.777$  ( $\tilde{J}(A, w) = 1.555, \tilde{J}(B, w) = 0.777$ ; the convergence point of TD(1) and a sub-optimal policy), it subsequently crossed into the optimal policy region, with  $w$  approaching  $-1$  ( $\tilde{J}(A, w) = -2, \tilde{J}(B, w) = -1$ ) within 30,000 iterations. The same result is achieved for every initial value of  $w$ . Further, we also ran this experiment with STD(1) observing two other policies: always choose  $a_2$  (the converse of the optimal policy), and randomly choose (with equal probability) between  $a_1$  and  $a_2$ . In both cases,  $w$  converged to the

same values as when observing the optimal policy. Hence, irrespective of which policy was being observed STD(1) learnt the correct ordering of the state values under the optimal policy.

### 5.1 Acrobot

In our second set of experiments, we applied both TD( $\lambda$ ) and STD( $\lambda$ ) to the much studied acrobot problem [Dejong and Spong, 1994; Sutton, 1996; Sutton and Barto, 1998]. This problem is analogous to a gymnast swinging on a high bar, and involves simulating a two-link underactuated robot. Torque can be applied only at the second joint. Our implementation is based upon the equations of motion and constants given in [Sutton and Barto, 1998, page 271]. However, angular velocities are restricted to the interval  $[-4\pi, 4\pi]$ , and both  $\ddot{\theta}_1$  and  $\ddot{\theta}_2$  are modified by adding in the damping term  $-\frac{|\dot{\theta}_m|}{\dot{\theta}_m} k \dot{\theta}_m^2$  where  $k$  is a constant and  $m$  is 1 or 2 as appropriate. The permitted actions are also restricted to applying torques of +1 or -1 at the second joint.

Actions are chosen every 0.1 simulated seconds, though motion is simulated at a much finer granularity. The system is run continuously with reward given after simulating the effect of each action; the reward being simply the height of the acrobot's tip above its lowest possible position.<sup>4</sup>

For our first experiment the learning algorithms observed a controller which used one-step look-ahead to greedily choose between two actions based on the following evaluation function.

$$J(\theta_1, \theta_2, \dot{\theta}_1, \dot{\theta}_2) = |\dot{\theta}_1 + \dot{\theta}_2| \quad (8)$$

This function was chosen because it implements a good policy which is significantly superior to its converse. Using a discount factor of  $\alpha = 0.95$ , we have empirically estimated the total discounted reward for the policy to be 24.4 (the maximum possible is 80, which requires the tip to be continuously balanced at the highest point), with an estimate for the converse being 0.4. The nature of our function approximator and feature vector limit the learning algorithms to choosing between the observed policy and its converse.

We used a linear function approximator, and a single element feature vector  $\phi(\cdot)$  (see (9) below) which is restricted to the range  $[0, 1]$  and gives high values to states *not preferred* by the hand-coded policy based on  $J(\cdot)$  (see (8) above).

$$\phi(\theta_1, \theta_2, \dot{\theta}_1, \dot{\theta}_2) = 1 - \frac{|\dot{\theta}_1 + \dot{\theta}_2|}{8\pi} \quad (9)$$

With the acrobot starting from rest in the vertical hanging position, and following the hand-coded policy mentioned above, the two learning algorithms were used to train separate linear evaluators. In both cases  $\lambda$  was set to 1.0. Note that since the observed policy is greedy, the probability of each state transition is 1; *i.e.* in the STD( $\lambda$ ) algorithm  $p^\mu(x_t, x_{t+1}) = 1$ .

TD( $\lambda$ ) converges to a weight of  $w = 27.3$ , whilst STD( $\lambda$ ) converges to  $w = -2.7$ . Since  $\phi(\cdot)$  orders states in the opposite order to  $J(\cdot)$ , a positive value for  $w$  means that TD( $\lambda$ ) has

<sup>4</sup>Both links are 1 metre in length, so the reward for each step is between 0 and 4.

converged to the converse of the hand-coded policy, *i.e.* the inferior policy.  $\text{STD}(\lambda)$  however, has converged to a negative  $w$  and thus its function approximator reverses the ordering imposed by  $\phi(\cdot)$  and implements the superior policy.

We subsequently repeated this experiment with the learning algorithms observing a random policy (actions chosen with equal probability, *i.e.*  $p^\mu(x_t, x_{t+1}) = 0.5$ ). Again  $\text{TD}(\lambda)$  converged to a positive  $w$  whilst  $\text{STD}(\lambda)$  converged to a negative  $w$ . Hence, irrespective of whether a good policy, or simply a random policy is being observed,  $\text{STD}(\lambda)$  learns a function approximator for this problem which implements a good set of relative state values (and hence a good policy), whilst  $\text{TD}(\lambda)$  learns an approximator whose relative state values generate an inferior policy.

## 6 Conclusion

We have shown that  $\text{TD}(\lambda)$  updates which seek to improve the state value approximation (by minimising  $\|\cdot\|_D$ ) can lead to inferior policies. For the system detailed in section 2,  $\text{TD}(\lambda)$  causes the function approximator to abandon an optimal policy. In section 3 we saw that in a real application, the  $\text{TD}(\lambda)$  updates can take the function approximator into regions of parameter space with policies inferior to those that had already been achieved.

For Binary Markov Decision Processes we presented a new algorithm,  $\text{STD}(\lambda)$ , that retains the advantages of  $\text{TD}(\lambda)$  in terms of on-line operation and small memory requirements (only the eligibility trace and current parameter vector need to be stored), but operates directly on the difference in values between *sibling states*, rather than the state values themselves. The limiting behaviour of  $\text{STD}(\lambda)$  was characterised for linear function approximators, yielding Theorem 1 covering its performance on two-state BMDPs, and an interpretation that  $\text{STD}(\lambda)$  acts to improve policies rather than the state values themselves. A comparison was made with Bertsekas'  $\text{DT}(\lambda)$ , and a three-state example presented which demonstrates the difference between  $\text{DT}(\lambda)$  and  $\text{STD}(\lambda)$ .

We have also successfully demonstrated that  $\text{STD}(\lambda)$  converges to optimal policies for the two-state system (consistent with Theorem 1) and the acrobot problem.

## References

[Baird, 1993] L.C. Baird. Advantage Updating. Technical Report WL-TR-93-1146, Wright Patterson AFB OH, 1993.

[Baxter *et al.*, 2000] Jonathan Baxter, Andrew Tridgell, and Lex Weaver. Learning to Play Chess Using Temporal-Differences. *Machine Learning*, 40(3):243–263, September 2000.

[Bertsekas, 1997] Dimitri P. Bertsekas. Differential Training of Rollout Policies. In *Proceedings of the 35th Allerton Conference on Communication, Control, and Computing*, Allerton Park, Ill., 1997.

[Dayan, 1992] P.D. Dayan. The convergence of  $\text{TD}(\lambda)$  for general  $\lambda$ . *Machine Learning*, 8:341–362, 1992.

[Dejong and Spong, 1994] G. Dejong and M. W. Spong. Swinging up the acrobot: An example of intelligent control. In *Proceedings of the American Control Conference*, pages 2158–2162, Evanston, IL, 1994.

[Gordon, 1995] G.J. Gordon. Stable function approximation in dynamic programming. Technical Report CMU-CS-95-103, Carnegie Mellon Univ., 1995.

[Harmon *et al.*, 1994] M.E. Harmon, L.C. Baird, and A.H. Klopf. Advantage Updating Applied to a Differential Game. Unpublished report presented at the 1994 Neural Information Processing Systems Conference, 1994.

[McGovern and Moss, 1998] Amy McGovern and Eliot Moss. Scheduling Straight-Line Code Using Reinforcement Learning and Rollouts. *Advances in Neural Information Processing (NIPS '98)*, 11, 1998.

[Samuel, 1959] A.L. Samuel. Some Studies in Machine Learning Using the Game of Checkers. *IBM Journal of Research and Development*, 3:210–229, 1959.

[Singh *et al.*, 1995] S.P. Singh, T. Jaakkola, and M.I. Jordan. Reinforcement learning with soft state aggregation. In G. Tesauro, D.S. Touretzky, and T.K. Leen, editors, *Advances in Neural Information Processing Systems*, volume 7. MIT Press, Cambridge, MA, 1995.

[Sutton and Barto, 1998] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge MA, 1998. ISBN 0-262-19398-1.

[Sutton, 1988] Richard Sutton. Learning to Predict by the Method of Temporal Differences. *Machine Learning*, 3:9–44, 1988.

[Sutton, 1996] Richard Sutton. Generalization in reinforcement learning: Successful examples using sparse coarse coding. In D. S. Touretzky, M. C. Mozer, and M. E. Hasselmo, editors, *Advances in Neural Information Processing Systems: Proceedings of the 1995 Conference*, pages 1038–1044. MIT Press, Cambridge MA, 1996.

[Tesauro, 1989] Gerald Tesauro. Connectionist Learning of Expert Preferences by Comparison Training. *Advances in Neural Information Processing*, 1:99–106, 1989.

[Tesauro, 1992] Gerald Tesauro. Practical Issues in Temporal Difference Learning. *Machine Learning*, 8:257–278, 1992.

[Tesauro, 1994] Gerald Tesauro. TD-Gammon, a self-teaching backgammon program, achieves master-level play. *Neural Computation*, 6:215–219, 1994.

[Tsitsikilis and Van Roy, 1997] John N Tsitsikilis and Benjamin Van Roy. An Analysis of Temporal Difference Learning with Function Approximation. *IEEE Transactions on Automatic Control*, 42(5):674–690, 1997.

[Utgoff and Clouse, 1991] Paul Utgoff and Jeffery Clouse. Two Kinds of Training Information for Evaluation Function Learning. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, pages 596–600. MIT Press, 1991.

[Utgoff and Saxena, 1987] Paul Utgoff and Sharad Saxena. Learning a Preference Predicate. In *Proceedings of the Fourth International Workshop on Machine Learning*, pages 115–121. Morgan Kaufmann, 1987.

[Werbos, 1990] Paul Werbos. A menu of designs of reinforcement learning over time. In W. T. Miller, R. Sutton, and P. Werbos, editors, *Neural Networks for Control*. MIT Press, Cambridge, MA, 1990.

[Werbos, 1992] Paul Werbos. Approximate Dynamic Programming for Real-Time Control and Neural Modeling. In D. A. White and D. A. Sofge, editors, *Handbook of Intelligent Control*. 1992.