# True Online TD(λ)

Sterling Suggs, Jonathan Dayton

March 7, 2018

## Introduction

Reinforcement learning describes a class of problems in which an agent learns to obtain some goal by learning through direct interaction with the environment. In other words, the agent is not told what to do at all; in many cases he does not even know what the goal is, but he receives a reward signal which he tries to maximize by exploring the state space and his available actions. A function mapping states to actions is called a policy; in these terms, the agent seeks to find the policy that will maximize reward.

There are two distinct parts of a general reinforcement learning task: determining the value of the current policy (called policy evaluation or *prediction*), and improving the policy for higher reward (called policy improvement or *control*). Control will be discussed later. A class of algorithms known as temporal-difference learning, or TD methods, solve the prediction problem by taking steps under the current policy, observing the reward, and then updating the prior estimate of the value of that action to align more closely with the reward actually received. Over many episodes of simulation, this algorithm will converge to the true state-value function for the given policy.

TD(λ) is one version of the general TD algorithm, that uses *eligibility traces* to further speed learning. The eligibility trace keeps track of how recently a state was visited, using this to assign credit for observed results to states that were visited more often or more recently. Advantages of this algorithm include ease of implementation; low complexity; and conceptual relationship to the theoretical forward view, in which the estimated value of each state is updated based on the total (discounted) return received thereafter [1].

However, TD(λ) is exactly equivalent to this forward view only if updates are postponed until the termination of each episode, since it is not till then that the total return is known. In practice, this is often impractical since tasks may not be episodic in nature, and even if they are, learning can be accelerated with more frequent updates. The online version of TD(λ), which updates the state-value function at each time step, is only approximately equivalent to the forward view.

## True Online TD(λ)

In the paper we chose to do our project on, published in 2014 in Proceedings of Machine Learning Research, Harm van Seijen and Richard Sutton propose a new version of the forward view, and a variant of TD(λ) which exactly matches it, even in the online case. Called true online TD(λ), this algorithm uses a new form of eligibility trace to update the value function in a way that exactly matches the forward view. Its complexity is the same as traditional TD(λ), and in the experiments of the authors, it demonstrated less error and faster learning.

In traditional TD(λ), decaying eligibility traces are used to approximate a forward-looking target called the λ-return, which is simply a weighted average of all n-step returns from the current state to the end of the episode [2]. True online TD(λ) modifies this by defining a truncated λ-return, which is similar to the usual λ-return, but only averages return estimates up to a certain time into the future. This allows a different formulation of the forward view, in which, at each time step, the truncated λ-returns from all previous time steps are updated and used to compute the current estimate of the state-value function.

This forward view is expensive to implement directly, since it requires keeping track of all observed states and rewards. Instead, Sutton and Seijen create a new form of eligibility trace, and use it to define an update rule that exactly matches this modified forward view. A thorough proof of this exact match for all time steps is included in the paper.

Finally, a word about function approximation. As stated before, the goal of TD(λ) is to learn a function to estimate the value of states. This function can be represented many ways. One common approach is a linear function approximator, which learns a set of weights *θ*, with which the value of state *s* is estimated as *θϕ(s)*, where *ϕ(s)* is the vector of features describing state *s*. This paper describes all parts of the algorithm in terms of linear function approximation, but any more general approximation could also be used.

We found this paper to be very clear and easy to follow. Our implementation of the algorithm succeeded with very little trouble.

## Experiments

We tested the true online TD(λ) algorithm in four tasks: two prediction tasks, and, for our creative experiments, two control tasks.

### Random Walk

The first prediction task is an 11-state linear random walk, with state 0 being terminal with reward 0, and state 11 being terminal with reward 1. The policy in this task is the probability of moving left or right at each state. For simplicity and ease of analysis we examined a simple policy of moving left or right at each state with equal probability. We found that TD(λ) quickly converged to values relatively near the truth, but tended to underestimate. Choice of parameters had a strong effect on convergence.

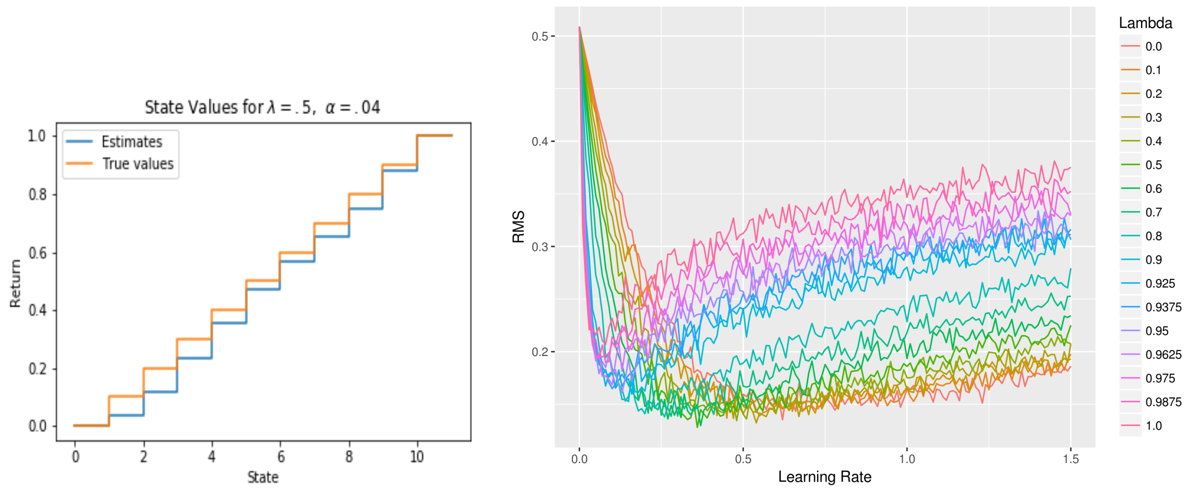
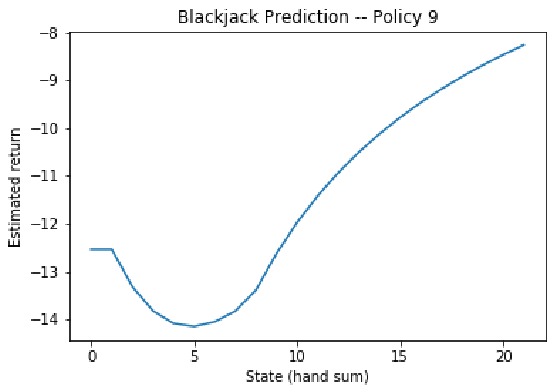


Figure 1: Random Walk: (**Left**) TD(λ) tends to underestimate the true state values. Some learning parameters give results that overestimate, but underestimation seems to be our norm. At this point we are not sure if this is a tendency of the algorithm itself, or something unique to our implementation. (**Right**) Root mean-squared error vs. learning rate for various values of λ. Note that the x-axis actually shows the learning rate multiplied by 10.

### Blackjack

Our second prediction task, one we invented ourselves, is an extremely simplified blackjack-type game, in which a single player draws cards one at a time from a deck. If the sum of his cards surpasses 21, he receives a reward of -10. Otherwise, he can choose when to stop drawing cards and take a reward equal to the sum of his cards minus 21 (a value at most 0). Our simulated deck considers all face cards to have value 10, and all others the value of their pips. Thus the expected value of a drawn card is 6.25.

The policy in this game is the choice of number at which, if the sum of his cards exceeds, the player will stop drawing more cards. As before, the value at any state (sum in hand) is the reward expected from that state to the end of the game, if the given policy is followed. We measured the state values for multiple policies and with the parameters $\lambda = .9$, with $\gamma = .65$ and $\alpha = .01$.



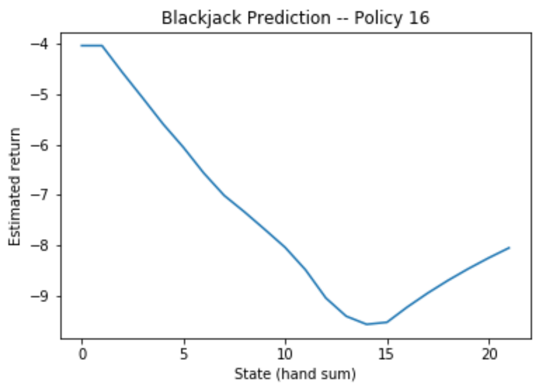


Figure 2: Blackjack state-value function for λ = .9, with $\gamma$ = .65 and $\alpha$ = .01. (**Left**) Policy to stop drawing cards when score surpasses 9. (**Right**) Policy to stop drawing cards when score surpasses 16.

One thing we learned, when using TD(λ) for prediction, is that, given the immense effect that parameter values can have on the result, one really has to know ahead of time what to look for. This can be a drawback in cases one does not have an idea of the expected result. Since we contrived this blackjack example ourselves, we were not sure what the true values should be, and we had to try many different parameters before getting a value function we could convince ourselves might be close to true.

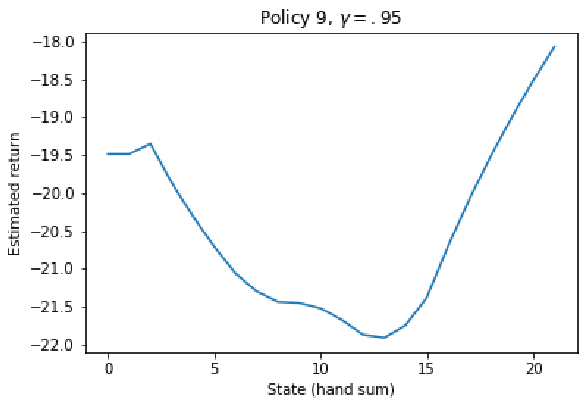


Figure 3: Predictions for policy 9, but with a discounting factor of .95, rather than .65 as in Figure 2.

Figure 3 shows an alternate state-value function for policy 9, with a higher discounting factor. We are skeptical of this, because under this policy, it is impossible to receive a reward lower than -12. This is a small concern in the former plot as well, but the discrepancy is smaller, and overall results with the former parameters better matched our expectations. This illustrates one weakness of the algorithm: because there is not one best set of parameters for all tasks, they must be tuned to match prior expectations which may or may not be valid. Certainly we could analyze the problem to discover the true state values, but this would defeat the whole purpose of using a learning algorithm.

## Creative Experimentation

As it stands, TD(λ) is an algorithm for prediction, that is, for policy evaluation. In most cases of interest, we desire not only to evaluate a given policy, but to update the policy for improved behavior. This is called *control*, and Seijen and Sutton state in their paper that TD(λ) can be "easily modified" for control, by using state-action features instead of state features, thus changing it to a true online Sarsa(λ) algorithm [1]. They claim that this version also performs better than ordinary Sarsa(λ), but they provide no implementation details.

Our creative experiment was to modify this algorithm for control, and test it on two tasks. The modifications turned out to be relatively straightforward, as the authors stated, by simply using state-action features instead of the state features of before. That is to say, the algorithm itself changed very little; most of the work lay in designing these state-action features. This work will be addressed momentarily.

An additional complication was the move, for our experiments, from a discrete to a continuous state space. Function approximation for discrete and continuous state spaces is similar, but the continuous case requires some method of generalization from a state to its neighbors, since any single state will almost surely be seen no more than once. A standard approach to this generalization is through coarse tile coding [3].

A coarse tiling is a partition of a continuous state space into discrete chunks. A feature for some given state is simply a one-hot vector indicating which chunk the state falls into. With a single tiling, this is equivalent to basic state aggregation. Better generalization can be achieved, however, through the use of multiple overlapping tilings. In this case, a feature for a given state is a "few-hot" vector that indicates which chunks, one per tiling, the state falls into. More detail can be found in [2] and [3].

We used coarse tile coding for our continuous-space control problems. The final task was to combine the state features provided by this tiling with the actions, to get state-action features that could be used by our new Sarsa(λ) algorithm. We were unable to find much research on the best way to do this. After several attempts with different methods, we settled on one that seemed effective. It essentially duplicated the weight vector $n$ times, where $n$ is the number of (discrete) actions: a state-action feature, then, was a vector of zeros of the same size as the weights, with the few-hot tile indicator vector sliced into the section corresponding to the action in question. In this way, separate weights could be learned for each action, and the values of each action could be estimated independently.

## Mountain Car

The first control task we attempted is the standard Mountain Car task. In this problem, a car, stuck in a valley, and too weak to drive straight out, must learn to build momentum by backing up the opposite wall. As discussed before, parameter tuning plays an essential role in getting these methods to learn. We found that for this task, a large discounting factor (greater than .95) was absolutely essential. Too, the learning rate could not be larger than .18 (see figure 4, right). As is standard procedure, actions were selected according to an epsilon-greedy policy; we found that it was helpful to start with a relatively large value of epsilon, $\varepsilon = 0.2$, and decay it exponentially down to about 0.03. The greater exploration early on, we felt, helped the car to find the goal faster.

## Pacman

Just for fun, we decided to throw the control version of TD(λ) at a much more complicated problem, the Atari game MsPacman. The state, as given by the simulator, was the 128-byte RAM of the Atari machine. Obviously this list of numbers is not very explicit about the location of Pacman, his enemies, the walls, or anything else useful. Without wanting to spend uncalled-for amounts of time designing a whole new algorithm, we just popped it into the algorithm two ways, with and without tile coding. Neither method proved effective, and we were not particularly surprised. A linear function approximator is probably not sufficient to learn the dynamics of a game as complicated as MsPacman; nor were our state features descriptive enough of the state of the game. We do not feel that this is a statement on the algorithm itself, but on the state features and function approximation we used, which are somewhat external. With a neural network to learn features and approximate the state-value function, TD(λ) may well have a chance.

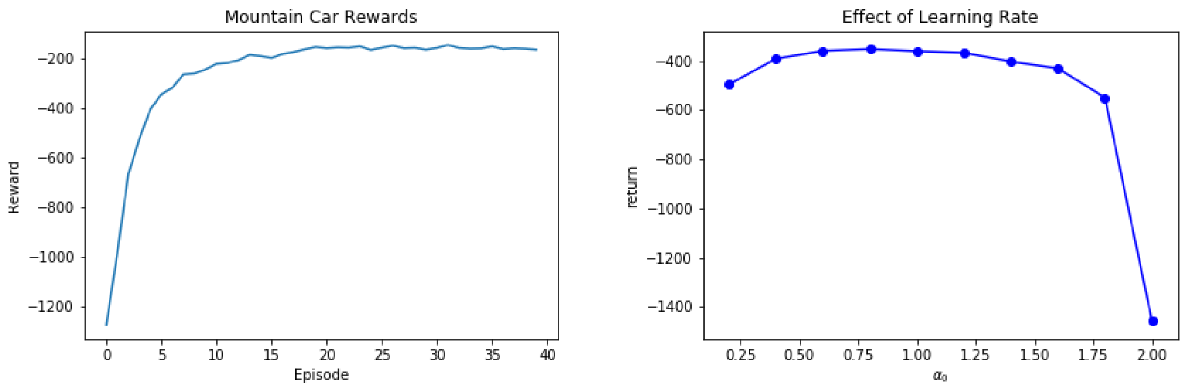


Figure 4: (**Left**) Mountain Car rewards vs. episodes, averaged over 50 runs. Note that after discovering the goal in the first couple episodes, learning proceeds quite rapidly. (**Right**) Average return over the first 20 episodes, averaged over 60 runs, as a function of learning rate. Note that the x-axis shows learning rate multiplied by 10.

## Summary and Analysis

The authors of the paper performed some of the same tasks that we did, comparing true online TD(λ) to the standard version in three benchmark tasks. In two different random-walk tasks, true online TD(λ) outperforms both the usual versions of traditional TD(λ) (with replacing and accumulating eligibility traces). To test the ability of this algorithm to generalize to control problems, they tested it on the mountain car task, where it again outperformed traditional TD(λ) in both its versions. We did not replicate these comparative experiments, because implementing regular TD(λ) was not part of our project.

A strength of this paper is the mathematical rigor applied by the authors, especially in including a full and detailed proof of their main claim, that this algorithm exactly matches the theoretical forward view. The proof is very easy to understand because the authors make sure to define every symbol that they use. They assume a minimum of background knowledge, going so far as to provide multiple pages explaining the usual approach to temporal-difference learning.

It could perhaps be strengthened further by more thorough or larger tests. Three toy examples do not seem like much ground to assert generally that the new version is "empirically" better than the traditional one.

One thing to be aware of is that the paper defines everything in terms that assume a linear function approximator. This is not crucial to the algorithm itself, which will work with more general function approximators such as neural nets. For a more complex task like Pacman, more work should be done either to manually create state features or a neural net to learn them.

Finally, as noted in the Blackjack section, the parameter tuning effectively turns any reinforcement learning problem into a reinforcement learning problem for the programmer as well. There is currently not much research into heuristics for parameter picking, with the result that the programmer must know what he is looking for to know if the algorithm is working. This is not so much the case in control tasks, where it is generally easier to observe if the agent's reward is increasing, but it still requires extensive parameter tuning that may, until one gains a sense of the problem, feel like stabbing blindly in the dark. However, this is often the case in much of machine learning, and we do not judge TD(λ) too harshly for it.

Overall, the true online TD(λ) algorithm is an excellent improvement over traditional TD(λ), itself a key algorithm in reinforcement learning. It provides a simple, efficient, and comprehensible method to learn optimal behavior in an unknown environment, and the applications of this are vast and relatively unexplored.

## References

[1] Seijen, H., Sutton, R.: True Online TD(λ). Proceedings of the 31st International Conference on Machine Learning, PMLR 32(1):692-700, 2014. (2014)

[2] Sutton, R., Barto, A. (1998). Reinforcement Learning: An Introduction. Cambridge, MA: MIT Press

[3] Sutton, R. (1996). Generalization in Reinforcement Learning: Successful Examples Using Sparse Coarse Coding. *Advances in Neural Information Processing Systems, 8*, 1038-1044.