Value Function Approximation with TD(λ) Learning

JONATHAN COLAÇO CARR, CESARE SPINOSO-DI PIANO, SABINA ELKINS
McGill University
February 24, 2022.

This work discusses a famous learning paradigm in reinforcement learning called TD(λ). While used extensively in practice, the theoretical properties of this learning method are not well understood for nonlinear function approximators. Highlighting the need to address this theory gap, we showcase the iconic performance of the TD-Gammon agent before reviewing the convergence properties of TD(λ) for linear function approximators. We emphasize the barriers that obscure our understanding of how TD(λ) interacts with non-linear value functions.

1. Introduction

Reinforcement learning (RL) is the set of problems and solutions related to maximizing cumulative reward over time in an interactive environment. Typically, this is presented as an *agent* interacting with an environment over discrete time steps[1]. At each time step, the agent takes an action A_t and observes a scalar reward R_t. Typically the agent chooses its action based on its current state S_t. By observing the *cumulative reward* from sequences of states and actions, the agent learns to better estimate which actions lead to higher expected future rewards for a given state.

RL agents can perform interactive tasks that are not well suited for supervised learning methods. This has been well documented in many game-playing environments, such as the game of backgammon. In section 3, we highlight the success of a TD(λ)-based agent, known as *TD-Gammon*. It one of the first successful applications of RL methods in which the learning agent was able to equal the level of play the top backgammon players. However, the theory behind TD-Gammon and TD(λ) learning remains unclear.

1.1. *Formalizing the RL Framework*

In this section we present the mathematical framework for RL. This will allow us to formally study RL algorithms and their convergence properties. To do so we introduce the Markov Decision Process (following [AJK19]) which underlies many RL problems.

Definition 1.1 A *Markov Decision Process* (MDP) is a tuple $M = (\mathcal{S}, \mathcal{A}, Q, R, \gamma, \mu)$ defined by

- A set of states $\mathcal{S}$, typically assumed finite our countably infinite,
- A set of actions $\mathcal{A}$, which may be continuous or discrete,
- A *transition function* $Q : \mathcal{S} \times \mathcal{A} \to \Delta(\mathcal{S})$, where $\Delta(S)$ is the space of probability distributions over $\mathcal{S}$. $Q(s'|s,a)$ is the probability of transitioning into the state s' upon taking action a in state s. As a Markovian process, the dynamics of the environment are fully described by Q.
- A *reward function* $R : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$,
- A discounting factor $\gamma \in [0,1]$,
- An initial state distribution μ over $\mathcal{S}$, specifying how the initial state S_0 is generated.

A typical MDP is interpreted as follows. After observing the initial state $S_0 \in \mathcal{S}$, the agent takes an initial action $A_0 \in \mathcal{A}(S_0)$ (where $\mathcal{A}(S)$ denotes the set of achievable actions from state S). This causes the environment to produce a scalar reward R_1 and transition the agent to a new state S_1. For *episodic tasks*, this process continues the agent reaches a *terminal state* $S \in \mathcal{T} \subseteq \mathcal{S}$[2]. This work only consider episodic tasks,

[1] For the purposes of our report, we restrict our discussion to discrete time step RL.
[2] The process continues indefinitely for *continuous tasks*.

since they describe game-playing RL agents.

To characterize the behaviour of the agent, we define the *policy* $\pi : \mathcal{A} \times \mathcal{S} \to [0,1]$ as the probability distribution of the actions the agent may take when in state s. It is denoted as

$$\pi(a|s) := P(A_t = a|S_t = s).$$

If π is deterministic, it can be redefined as a mapping of states to actions, i.e. $\pi : \mathcal{S} \to \mathcal{A}$.

1.2. *Learning in RL*

Recall that the goal of an RL agent behaving in an MDP $M = (\mathcal{S}, \mathcal{A}, Q, R, \gamma, \mu)$ is to maximize its cumulative reward. We define the cumulative reward starting from time step t as

$$G_t := R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots + \gamma^{T-t-1} R_T = \sum_{k=1}^{T-t} \gamma^{k-1} R_{t+k}.$$

The discount factor γ acts as a way of mitigating immediate and future reward. For continuous tasks, it also ensures that G_t is bounded whenever the rewards are bounded. Given that the environment (and policy) may be stochastic, the goal of the RL agent is to try and learn the expected cumulative reward for each state. This estimate is given by a *value function* $v_\pi : \mathcal{S} \to \mathbb{R}$, defined as

$$v_\pi(s) = \mathbb{E}_\pi[G_0|S_0 = s].$$

Note that $v_\pi(s)$ depends on the behaviour policy π. Thus, given a state s, the goal of a RL agent is to find a way of behaving π that maximizes the value $v_\pi(s)$. That is, we would like to find a π^* such that

$$\pi^* \in \text{argmax}_{\pi \in \Pi} v_\pi(s),$$

where Π is the set of all possible policies or ways of behaving. In most realistic settings v_π is unknown. This is either because: (1) Q is unknown[3] or (2) if Q is known, the size of the state space makes computing $v_\pi(s)$ exactly intractable. As a result, we are required to find *estimates* of $v_\pi(s)$, denoted $V_\pi(s)$, based on the previous observations of the agent. *Temporal-difference learning* algorithms are one of the most common strategies used to compute these estimates.

2. Temporal-Difference Learning

In RL learning algorithms, estimates of the value function v_π are typically updated using an incremental update of the form

$$NewEstimate \leftarrow OldEstimate + StepSize * (Target - OldEstimate).$$

In our case, we update $V(s)$ as

$$V(s) \leftarrow V(s) + \alpha * (Target - V(s)) \quad \forall s \in \mathcal{S}.$$

Temporal-difference (TD) learning is characterized by the *Target* used to update estimates. It is inspired by the following recursive property of the true value function for a given policy v_π, first observed by Bellman

[3] A result from [Put14] showed that restricting our search space to deterministic policies was enough so the only stochastic component that is left is Q.

[Bel57].

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi[G_t|S_t = s] && \text{Generalized definition of the value function} \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+t}|S_t = s] && \text{Writing } G_t \text{ recursively} \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1})|S_t = s] && \text{Using total expectation and the Markov property} \end{aligned}$$

The intuition behind the *Target* in temporal-difference learning is to emulate the term $R_{t+1} + \gamma v_\pi(S_{t+1})$ by replacing the true value function v_π with its current estimate V. This approach, known as *bootstrapping*, allows agents to learn surprisingly quickly despite using a target which is biased towards current estimates [SB18].

Example 2.1 The simplest example of temporal-difference learning algorithm is *TD(0)*, which generates trajectories under policy π and updates its value estimate as follows:

$$V(S_t) \leftarrow V(S_t) + \alpha * (R_{t+1} + \gamma V(S_{t+1}) - V(S_t)).$$

The term $R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$ is also referred to as the *TD error*.

2.1. *Value Function Approximation*

So far we have implicitly dealt with estimates of the value function V which are entirely tabular. That is, each state $s \in \mathcal{S}$ has a unique entry in V which is updated independently of other estimates. This quickly becomes infeasible for more realistic settings such as backgammon where the state spaces do not fit into memory (backgammon has an estimated 10^{20} possible state configurations). Moreover, even when tabular methods are possible they are not always the most efficient approach. This is because states which may have the same true value are estimated separately by V.

Value function approximation overcomes these challenges by using a parameterized function $V(S) = V(S;w)$. Instead of updating our estimate for each *state*, we now only update our *parameterization* w. To do so we translate the "typical" RL update rule into the gradient-based update:

$$NewEstimate \leftarrow OldEstimate - StepSize * NormalizingFactor * \nabla Error$$

For TD(0) value function approximation, we use the squared TD(0) error as the $Error$ (and consequently a $NormalizingFactor$ of $\frac{1}{2}$) to get the following update for w:

$$\begin{aligned} w &\leftarrow w - \frac{1}{2}\alpha\nabla_w(R_{t+1} + V(S_{t+1}) - V(S_t))^2 \\ &\leftarrow w - \alpha(R_{t+1} + V(S_{t+1}) - V(S_t))\nabla_w(R_{t+1} + V(S_{t+1}) - V(S_t)) && \text{Chain rule} \\ &\leftarrow w + \alpha(R_{t+1} + V(S_{t+1}) - V(S_t))\nabla_w V(S_t) && \text{Assuming fixed target} \end{aligned}$$

The last assignment gives us the form of the semi-gradient function approximation based TD(0) learning algorithm. This is a semi-gradient algorithm because the gradient computed with respect to the error is not a true error since TD(0) makes the simplifying assumption that its target is fixed with respect to w.

Example 2.2 One simple value function approximator is the *linear function approximator*. With each state S encoded as a feature vector x_S, the estimated valued function of S is $V(S;w) = w \cdot x_S$. In this case

$$\nabla_w V(S) = \nabla_w w^T x_S = x_S,$$

and thus the semi-gradient TD(0) update has the form

$$w \leftarrow w + \alpha(R_{t+1} + V(S_{t+1}) - V(S_t)) \cdot x_S.$$

2.2. *TD(λ)*

TD(0) updates only look at the immediate next reward when updating the value function. However, it is possible to extend TD(0) and incorporate experience from previous time steps. It may also be beneficial to down-weight these time-delayed observations, since intuitively we expect that they are less important for our immediate state and action.

The TD(λ) learning method adopts this strategy, considering all previous states with a recency weight-decay approach. Using the hyperparameter $\lambda \in [0,1]$, TD(λ) adjusts its sensitivity to the most recent states. For value function approximation, TD(λ) methods apply the following update rule:

$$w_{t+1} \leftarrow w_t + \alpha(V(S_{t+1}) - V(S_t))(\sum_{k=1}^{t} \lambda^{t-k} \nabla_w V(S_k)), \tag{2.1}$$

This is the same as the update in TD(0) with the addition of the recency-weighted sum of the previous states.

The choice of λ can be understood in terms of a bias-variance tradeoff. As $\lambda \to 1$, previously observed states will have a larger influence on the weight update. This effectively de-biases our model from its previous estimate. As $\lambda \to 0$, the previously observed states becomes less important. Instead the difference in value estimates of our current value function has more influence. This effectively reduces our exposure to the variance of the current trajectory.

3. TD-Gammon

We will now explore a famous application of temporal-difference learning: TD-Gammon. TD-Gammon is a backgammon playing algorithm created by Gerald Tesauro in the 90s [Tes95]. This algorithm was one of the first popular game-playing RL agents that was able to learn by playing against itself. This caused great excitement in the field at the time, and it still relevant to the foundation of RL theory today [SB18].

3.1. *Game Playing Algorithms*

Game playing algorithms have had a massive role in the development of the field of reinforcement learning [Sch99]. Their emergence in the later half of the 20th century can be attributed to the fact that board games can easily be modeled as a finite, fully observable, discrete time Markov decision process. This property makes them excellent for studying whether or not RL algorithms can succeed on difficult tasks, without having to make limiting assumptions about the problem at hand or have a overly complicated representation of the environment.

The first RL game playing algorithm to receive public recognition was Samuel's Checker's Player in the 60s [Sam67]. This algorithm learned to play the game of checkers by playing thousands of games against itself. This technique became central to many problems in RL. In the 60s, this had never been tried before. The algorithm's first version used TD learning with a linear function to approximate the value of the different game states. A later version implemented a simple neural network for its value estimates. Samuel's Checker's Player was able to learn a good strategy for the game and compete at a beginner to intermediate human-player level [SW10], which was an unprecedented, and hugely exciting achievement!

The next big board game playing algorithm that became famous is the focus of this whole section: TD-Gammon. Building on Samuel's work from the 60s, Tesauro applied TD learning to a much more complex board game. The complexity of the game, along with the lack of background knowledge given to the algorithm, is what made TD-Gammon's success at Backgammon so incredible at the time [SB18].

3.2. *TD-Gammon Structure*

The game of Backgammon is a two-player board game with a lot of complexity, despite being played on a one-directional, one-dimensional track. Setting the rules and strategy of Backgammon aside, note that it is possible to formulate this game as a Markov decision process with positive or negative rewards R for winning or losing the game. Each possible state of the board is a state $S \in \mathcal{S}$. The state space is massive, containing approximately 10^{20} states, which makes this task impossible to complete with tabular approaches. The action space is also enormous, considering that from each of the 10^{20} possible states, given a random dice roll, it is possible to move to around 20 different states [Tes95].

3.2.1. Value Function and Policy

TD-Gammon's value function V is a multi-layer perceptron (MLP). This is a huge jump from the linear value functions that were grounded in TD theory (see section 2). The theoretical implications of this transition to nonlinear value functions will be addressed in section 4.

The TD-Gammon value function contains 198 input units representing the state of the board. In some versions, additional input units with handcrafted features help evaluate the advantage of a given state. The network also contains 40 or 80 hidden units (again, depending on which version of the algorithm is considered). Finally, the network has an output that gives the predicted probability of winning from the input state (i.e. the *value* of the state). TD-Gammon's output is four units: two representing the probability of either player winning, and two representing the probability of either player winning with a "gammon" (i.e. a special way to win the game with more points). There is a third possible outcome of a game, called a backgammon, which earns the winner even more points. This outcome is rare, so Tesauro chose to exclude it [Tes95]. As it plays against itself during training, TD-Gammon improves the weights of its value function. The weight updates occur at every step of the game play, as explained in section 3.2.2.

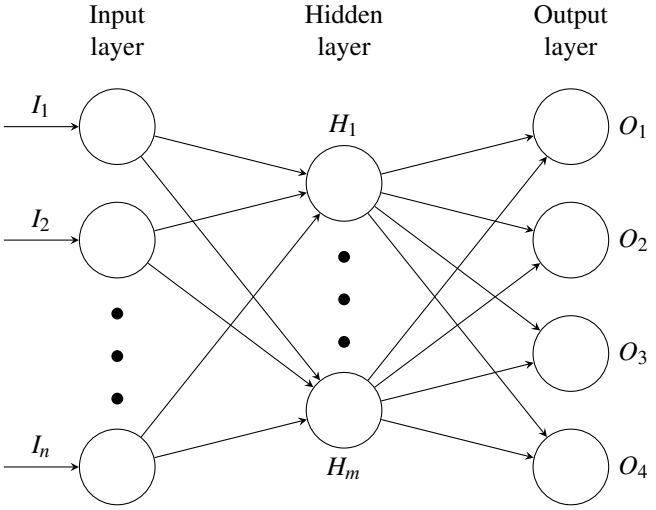

Figure 1. The TD-Gammon Value Function, where $I_1, ..., I_n$ are the inputs to the function (198 representing the board state, plus the handcrafted features), $H_1, ..., H_m$ are the hidden layers, and $O_1, ..., O_4$ are the outputs (i.e. what the value function returns for a given state).

TD-Gammon's policy, π, is a greedy policy based on the value function described above. It scores all legal moves at each time step of the game and chooses that which maximizes the expected outcome. The computation of this is relatively fast, since given a random dice roll and the current state, there are about 20 possible moves from any given state in backgammon.

3.2.2. TD-Gammon's Pseudocode

Algorithm 1 Simplified Pseudocode for TD-Gammon

Inputs: policy π, learning rate $\alpha \in (0,1]$
Initialize value function $V(S) \forall\, S \in \mathcal{S}$ randomly
loop for each episode
  Initialize S_1
  loop for every step of episode
    Take $a = \pi(S_t)$, observe R and S_{t+1}
    $w_{t+1} \leftarrow w_t + \alpha(V(S_{t+1}) - V(S_t))(\sum_{k=1}^{t} \lambda^{t-k} \nabla_w V(S_k))$
  end loop when S_t is terminal
end loop

Algorithm 1 shows a simplified version of the steps followed by TD-Gammon during training. To start, the algorithm needs an input policy π, as explained in section 3.2.1, and a value for the learning rate, α. The algorithm will start off by initializing the weights of its value function randomly.

For every training episode (i.e. a full game), the algorithm sets S_1 as the starting position of backgammon. Then, for every time step in the episode, the algorithm uses its policy to chose and take an action. TD-Gammon then observes the reward (which is mostly 0 unless S_t is a terminal state) and the next state, S_{t+1}. At this point, TD-Gammon will perform the TD(λ) weight update from Eq 2.1.

3.3. *Results*

Competing against backgammon world champions in the 90s, TD-Gammon held its own. Tesauro reports three different versions of TD-Gammon and their scores against the human world champions: TDG 1.0, TDG 2.0, and TDG 2.1 [Tes95]. The first version, TDG 1.0, had 80 hidden units but included no additional hand-crafted features for V. Without any knowledge outside of the current state and the weights learned through experience, this version trailed behind its human competition by only 13 points in 51 games [Tes95]. For TDG 2.0, Tesauro increased the number of training games, added in handcrafted features, but decreased the number of hidden units to 40. This agent trailed behind the world champion opponents by only 7 points in 38 games [Tes95]. Finally, Tesauro increased the number of training games once more and changed the number of hidden units back to 80, creating TDG 2.1. Over 40 games, it trailed just 1 point behind its world-class opponent [Tes95]. These scores against human world champion backgammon players were unprecedented at the time.

Beyond these exciting scores, TD-Gammon had other massive impacts in both the field of reinforcement learning and the world of backgammon. In backgammon, some results from TD-Gammon changed the way experts play certain difficult-choice scenarios, such as switching from 'slotting' to a 'split play' with an opening dice roll of 2-1, 4-1, or 5-1 [Tes95]. This new strategy performed better in practice than 'slotting' and is now widely used amongst backgammon masters. In reinforcement learning, TD-Gammon shed light on the combination of TD learning and MLPs. For instance, TD-Gammon emphasized the difference between absolute and relative errors in function estimation. While its value function was often off by a tenth, it was still consistently able to choose the highest value next move, since the entire function was off by this margin. TD-Gammon also posited that neural networks learn linear concepts first during training through the analysis of it's training games [Tes95]. These early concepts were a great initial strategy, and perform better than human beginner strategies. Its definitive success proved that TD-Gammon

was an effective solution for playing backgammon and helped advance the field of RL towards what we know today.

4. TD(λ) Convergence Properties

While TD(λ) learning has been tremendously successful in practice, relatively little has been shown about its theoretical properties. Linear value function approximators (described in Example 2.2) are known to converge to the true value functions under TD(λ) updates. However, there is no analogous result for *any* class of non-linear function approximator. In this section we review convergence in for linear approximators before addressing the theory gap to the non-linear case.

4.1. *Linear TD(λ)*

The TD(λ) learning method was formalized over 30 years ago [Sut88] and was quickly shown to produce correct value function estimates for linear function approximators [Sut88, Day92, DS94]. In the original framework, a sequence of states $(S_t)_{t=1}^{T}$ is observed for each episode, followed by an outcome (e.g. discounted cumulative return) G. The goal of TD(λ) is to update the value estimates of G, denoted $V^{(n)}(S)$, at each episode n and ensure they converge to $\mathbb{E}[G|S]$. Dayan [Day92, DS94] shows that these value estimates converge in expectation (and with probability 1) to the true values for linear value functions. The formal statement is given below.

Theorem 4.1 For any absorbing Markov chain and any starting probabilities μ_i with no inaccessible states, for any outcome distributions G with finite expectation, and for any linearly independent set of observation vectors $\{x_S : S \in \mathcal{S}\}$, the predictions of a linear value function converge in expectation to the ideal predictions under TD(λ) updates for $\lambda \in [0,1]$. I.e. for any $S \in \mathcal{S}$,

$$\lim_{n\to\infty} V^{(n)}(S) = \lim_{n\to\infty} \mathbb{E}\left[w^{(n)} \cdot x_S\right] = \mathbb{E}\left[G|S\right]. \tag{4.1}$$

Here we identify the key points in the proof of Theorem 4.1 where a linear value function is necessary[4]. This will help identify the theoretical shortcomings in the non-linear case.

Before proving convergence, Dayan notes that for absorbing Markov chains, we can rewrite $\mathbb{E}[G|S]$. Letting $Q = [q_{ij}]$ be the matrix of transition probabilities from S_i to S_j, define

$$\mathbb{R}^{|\mathcal{S}|} \ni h = \sum_{S_j \in \mathcal{T}} q_{ij}\mathbb{E}[G|S_j],$$

to be the vector where each vector component $[h]_i$ is expected value of G if S_i is the last non-terminal state in the episode. Dayan shows that for any state $S_i \in \mathcal{S}$,

$$\mathbb{E}[G|S_i] = [(I-Q)^{-1}h]_i. \tag{4.2}$$

The first advantage of linear value functions is that their weights can be directly interpreted as value estimates. From Eq 4.2, the statement of the theorem reduces to showing that

$$\lim_{n\to\infty} \mathbb{E}[w^{(n)}] \cdot x_{S_i} = [(I-Q)^{-1}h]_i \quad \forall S_i \in \mathcal{S}.$$

Since we assume that our states can be represented by linearly independent vectors, no generality is lost if we use one-hot vectors (i.e. the standard basis) to represent our states. When x_{S_i} is a one-hot encoding,

[4] see appendix A for full proof.

the previous equation is equivalent to saying that the weight vector of the linear value function converges component-wise to the true value function. So it remains to show that

$$\lim_{n\to\infty} \mathbb{E}[w^{(n)}] = (I-Q)^{-1}h.$$

Recall that the behaviour of $(w^{(n)})_{n\geq 0}$ is entirely determined the TD(λ) updates from Eq. 2.1. Thus, we need only examine the individual TD weight updates to prove that the value estimates converge. This is fundamentally different for non-linear function approximators, where individual weight updates do not uniquely determine the final value estimate.

The second key simplification for linear TD(λ) involves the weight update equation. Recall from Algorithm 1 that after the n-th episode, the following update occurs:

$$w^{(n+1)} \leftarrow w^{(n)} + \sum_{t=1}^{T} \alpha(V^{(n)}(S_{t+1}) - V^{(n)}(S_t)) \left(\sum_{k=1}^{t} \lambda^{t-k} \nabla_{w^{(n)}} V^{(n)}(S_k) \right), \tag{4.3}$$

where $S_{T+1} := G$. Since our states are one-hot encodings, the gradient of our linear function approximator essentially counts the number of times we visit state S, weighted with exponential decay:

$$\begin{aligned} \nabla_{w^{(n)}} V^{(n)}(S) &= \nabla_{w^{(n)}} (w^{(n)} \cdot x_S) = x_S \\ \Rightarrow \sum_{k=1}^{t} \lambda^{t-k} \nabla_{w^{(n)}} V^{(n)}(S_k) &= \sum_{k=1}^{t} \lambda^{t-k} x_{S_k}, \end{aligned}$$

So each component of the weight vector in Eq 4.3 is updated as

$$[w^{(n+1)}]_i \leftarrow [w^{(n)}]_i + \sum_{t=1}^{T} \alpha(w^{(n)}(x_{S_{t+1}} - x_{S_t})) \left(\sum_{k=1}^{t} \lambda^{t-k} \mathbf{1}_{S_i}(S_k) \right), \tag{4.4}$$

where

$$\mathbf{1}_{S_i}(S_k) = \begin{cases} 1 & S_i = S_k, \\ 0 & \text{otherwise.} \end{cases}$$

Taking an expectation over the weight vectors, the indicator function allows us to recover the expected number of times that we visit each state in a trajectory of length t.

With the weight updates in the form of Eq 4.4, Dayan takes the expectation of the weight vectors and re-groups the summands to show that they converge to the true values. While the rest of the proof is very technical, it does not depend on the linearity of the value function.

4.2. *Non-Linear TD(λ)*

Unfortunately the convergence of the TD(λ) is not guaranteed for nonlinear function approximators. Tsitsiklis and Van Roy [TVR97] describe an example where a non-linear value function diverges under TD(λ) weight updates. Thus, we know that certain assumptions must be made about the structure of the value function to in order to prove convergence.

Even when we do consider special cases (for example, two layer MLP used for TD-Gammon), theoretical properties of TD(λ) updates are difficult to recover. From our study of Dayan's proof section 4.1, we identify two possible reasons for this. The first is that the gradient term becomes more complicated. It

no longer behaves like a 'state counting' mechanism.

The second challenge is that the individual weight updates cannot be directly interpreted as value function estimates. This is particularly problematic for modern neural network architectures, where potentially billions of weights interact with each other to produce a single value estimate. The relationship between individual parameters and final predictions in these networks not well known. We believe that this is the most significant challenge in generalizing Dayan's proof to the non-linear setting.

5. Summary

The TD(λ) learning algorithm is a fascinating object to study. On the one hand, it provides almost no theoretical guarantees for most function approximators. Existing proofs for linear function approximators do not generalize well to the non-linear setting. On the other hand, many successful RL agents update their non-linear value functions with TD(λ) methods and achieve incredible results. Even the "simple" multi-layer perceptron used in TD-Gammon redefined what was thought possible for artificial intelligence.

Many open questions about the TD(λ) algorithm point directly to the intersection of reinforcement learning and neural network optimization. A better understanding of TD(λ) convergence may lead to a deeper understanding of both of these research areas.

REFERENCES

AJK19. Alekh Agarwal, Nan Jiang, and Sham M. Kakade. Reinforcement learning: Theory and algorithms. 2019.

Bel57. Richard Bellman. *Dynamic Programming*. Dover Publications, 1957.

Day92. Peter Dayan. The convergence of td(λ) for general λ. *Machine Learning*, 8:341–362, 1992.

DS94. Peter Dayan and Terrence J. Sejnowski. Td(λ) converges with probability 1. 1994.

Put14. Martin L Puterman. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014.

Sam67. Arthur L Samuel. Some studies in machine learning using the game of checkers. ii—recent progress. *IBM Journal of research and development*, 11(6):601–617, 1967.

SB18. Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018.

Sch99. Jonathan Schaeffer. The role of games in understanding computational intelligence. *IEEE Intelligent Systems*, pages 10–11, November/December 1999.

Sut88. Richard Sutton. Learning to predict by the method of temporal differences. *Machine Learning*, 3:9–44, 08 1988.

SW10. Claude Sammut and Geoffrey I. Webb, editors. *Samuel's Checkers Player*, pages 881–881. Springer US, Boston, MA, 2010.

Tes95. Gerald Tesauro. Temporal difference learning and td-gammon. *Commun. ACM*, 38(3):58–68, March 1995.

TVR97. J.N. Tsitsiklis and B. Van Roy. An analysis of temporal-difference learning with function approximation. *IEEE Transactions on Automatic Control*, 42(5):674–690, 1997.

WD89. Christopher J. C. H. Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8(3):279–292, May 1989.

A. Proof of TD(λ) Convergence

We let $V^{(n)}$ be the value function after the n-th episode of training. Dayan's proof of convergence [Day92] relies on a representation of the value function under TD(λ) updates, originally proposed by Watkins [WD89]. For a trajectory $(S_t)_{t=1}^{T}$, Watkins defined the *r step estimates* of the initial state S_0,

$$V_r^{(n)}(S_0) = \begin{cases} V^{(n)}(S_r) & S_r \in \mathcal{N} := \{\text{non-terminal states}\}, \\ G & \text{otherwise}, \end{cases}$$

Watkins showed that

$$V^{(n)}(S) = (1-\lambda)\sum_{r=1}^{\infty} \lambda^{r-1} V_r^{(n)}(S).$$

For linear value function approximators this means that

$$w^{(n+1)} = (1-\lambda)\sum_{r=1}^{\infty} \lambda^{r-1} w_r^{(n+1)}. \tag{A.1}$$

The advantage of r step estimates is that they are better than the $V^{(n)}$ estimates (in expectation). This is because they include feedback from G. Our aim will be to work with the weights $w_r^{(n)}$, and then use Eq A.1 to show our value function weights converge to the true values.

Under the TD(λ) learning updates and Eq A.1, the following update occurs after each training episode:

$$\begin{aligned} w_r^{(n+1)} &\leftarrow w_r^{(n)} \sum_{S \text{ visited}} [V_r^{(n)}(S) - V^{(n)}(i)] \nabla_{w^{(n)}} V^{(n)}(i) \\ &\leftarrow w_r^{(n)} + \alpha \sum_{S \text{ visited}} [V_r^{(n)}(i) - w^{(n)} \cdot x_S] x_S. \end{aligned} \tag{$\star$}$$

Define $\eta_{ij}^{(s)}$ as the number of times the s-step transition

$$x_i \to x_{k_1} \to \cdots \to x_{k_{s-1}} \to x_j,$$

occurs, for any intermediate states $x_{k_r} \in \mathcal{N}$. Then $(\star)$ can be regrouped by source and destination states of the transitions:

$$\begin{aligned} w_r^{(n+1)} \leftarrow w_r^{(n)} &+ \alpha \sum_{i\in\mathcal{N}} \sum_{j_r\in\mathcal{N}} \eta_{ij_r}^{r} [w_r^{(n)} \cdot x_{j_r} - w_r^{(n)} \cdot x_i] x_i \\ &+ \alpha \sum_{i\in\mathcal{N}} \sum_{j_r\in\mathcal{T}} \eta_{ij_r}^{r} [G_{j_r} - w_r^{(n)} \cdot x_i] x_i \\ &+ \alpha \sum_{i\in\mathcal{N}} \sum_{j_{r-1}\in\mathcal{T}} \eta_{ij_{r-1}}^{r-1} [G_{j_{r-1}} - w_r^{(n)} \cdot x_i] x_i \\ &\vdots \\ &+ \alpha \sum_{i\in\mathcal{N}} \sum_{j_1\in\mathcal{T}} \eta_{ij_1}^{1} [G_{j_1} - w_r^{(n)} \cdot x_i] x_i. \end{aligned} \tag{$\dagger$}$$

Here G_j indicates that the terminal value is generated from the distribution due to state j. The extra terms are generated by the possibility that, from visiting any $x_i \in \mathcal{N}$, the chain absorbs before taking r further steps. Taking expected values over sequences, for $i \in \mathcal{N}$, the following holds:

$$\begin{aligned} \mathbb{E}[\eta_{ij}^{r}] &= d_i Q_{ij}^{r} \quad \forall j \in \mathcal{N} \\ \mathbb{E}[\eta_{ij}^{s}] &= \sum_{k\in\mathcal{N}} d_i Q_{ij}^{s} q_{kj} \quad \forall j \in \mathcal{T} \text{ and } 1 \le s \le r. \end{aligned}$$

Here d_i is the expected number of times the Markov chain is in state i in one sequence. For an absorbing Markov chain, it is known that

$$d_i = \sum_{j\in\mathcal{N}} \mu_j (I-Q)^{-1}_{ji} = [\mu^T (I-Q)^{-1}]_i.$$

We are now ready to express $(\dagger)$ in terms of transition probabilities. Letting $\overline{w}$ be the expected values and noting that the dependence of $\mathbb{E}[w_r^{(n+1)} | w_r^{(n)}]$ on $w_r^{(n)}$ is linear,

$$\begin{aligned} \overline{w}_r^{(n+1)} \leftarrow \overline{w}_r^{(n)} + \alpha \sum_{i\in\mathcal{N}} d_i x_i \Bigg[&\sum_{j_r\in\mathcal{N}} Q_{ij_r}^{n} (x_{j_r} \cdot \overline{w}_r^{(n)}) \\ &- (x_i \cdot \overline{w}_r^{(n)}) \Bigg[\sum_{j_r\in\mathcal{N}} Q_{ij_r}^{n} + \sum_{j_r\in\mathcal{T},k\in\mathcal{N}} Q_{ik}^{r-1} q_{kj_r} + \cdots + \sum_{j_1\in\mathcal{T}} q_{ij_1} \Bigg] \\ &+ \sum_{j_r\in\mathcal{T},k\in\mathcal{N}} Q_{ij}^{r-1} q_{kj_r} \overline{G}_{j_r} + \sum_{j_{r-1}\in\mathcal{T},k\in\mathcal{N}} Q_{ij}^{r-2} q_{kj_{r-1}} \overline{G}_{j_{r-1}} + \cdots + \sum_{j_1\in\mathcal{T}} q_{ij_1} \overline{G}_{j_1} \Bigg]. \end{aligned}$$

Next, define X to be the matrix whose columns are x_i, so $[X]_{ab} = [x_a]_b$, and D to be the diagonal matrix $[D]_{ab} = \delta_{ab} d_a$, where δ_{ab} is the Kronecker delta. Recalling that $h_i = \sum_{j\in\mathcal{T}} q_{ij} \overline{G}_j$, we can convert the weight

updates to the matrix form

$$\overline{w}_r^{(n+1)} = \overline{w}_r^{(n)} + \alpha XD[Q^r X^T \overline{w}_r^{(n)} - X^T \overline{w}_r^{(n)} + (Q^{r-1} + Q^{r-2} + \cdots + I)h]. \qquad (\star\star)$$

We've used the fact that

$$\sum_{j_r \in \mathcal{N}} Q^r_{ij_r} + \sum_{j_r \in \mathcal{T}, k \in \mathcal{N}} Q^{r-1}_{ij} q_{kj_r} + \cdots + \sum_{j_1 \in \mathcal{T}} q_{ij_1} = 1,$$

as this covers all the possible options for r-step moves from state i. Lastly, we define the true values $[\overline{e}]_i = \mathbb{E}[G|S_i]$. Then

$$\begin{aligned}
\overline{e} &= [\mathbb{E}[G|S]] \\
&= h + Qh + Q^2 h + \cdots \\
&= (I + Q + Q^2 + \cdots + Q^{r-1})h + Q^r(I + Q + Q^2 + \cdots + Q^{r-1})h + \cdots \\
&= \sum_{k=0}^{\infty} [Q^r]^k (I + Q + Q^2 + \cdots + Q^{r-1})h \\
&= (I - Q^r)^{-1}(I + Q + Q^2 + \cdots + Q^{r-1})h,
\end{aligned}$$

where the sum converges since the chain is absorbing. This is another way of writing $\mathbb{E}[G|S_0]$. Multipliying $(\star\star)$ on the left by X^T,

$$\begin{aligned}
X^T \overline{w}_r^{(n+1)} &= X^T \overline{w}_r^{(n)} + \alpha X^T XD[(I + Q + Q^2 + \cdots + Q^{r-1})h + Q^r X^T \overline{w}_r^{(n)} - X^T \overline{w}_r^{(n)}] \\
&= [I - \alpha X^T XD(I - Q^r)] X^T \overline{w}_r^{(n)} + \alpha X^T XD(I + Q + Q^2 + \cdots + Q^{r-1})h.
\end{aligned}$$

Subtracting $\overline{e}$ from both sides of the equation and noting that $(I - Q^r)\overline{e} = (I + Q + Q^2 + \cdots + Q^{r-1})h$, this gives the update rule, which is the equivalent of the difference between true value and predictions:

$$\begin{aligned}
[X^T \overline{w}_r^{(n+1)} - \overline{e}] &= [I - \alpha X^T XD(I - Q^r)] X^T \overline{w}_r^{(n)} + \alpha X^T XD(I - Q^r)\overline{e} - \overline{e} \\
&= [I - \alpha X^T XD(I - Q^r)][X^T \overline{w}_r^{(n)} - \overline{e}].
\end{aligned}$$

Since $(1-\lambda)\sum_{r=1}^{\infty} \lambda^{r-1} = 1$ for $0 < \lambda < 1$, Eq. A.1 shows that

$$\begin{aligned}
[X^T \overline{w}^{(n+1)} - \overline{e}] &= \left\{ (1-\lambda) \sum_{r=1}^{\infty} \lambda^{r-1} [I - \alpha X^T XD(I - Q^r)] \right\} [X^T \overline{w}^{(n)} - e] \\
&= \left\{ I - \alpha X^T XD(I - (1-\lambda)Q[I - \lambda Q]^{-1}) \right\} [X^T \overline{w}^{(n)} - \overline{e}],
\end{aligned}$$

where $\overline{w}$ are the expected weights from the TD(λ) procedure. The sum

$$(1-\lambda) \sum_{r=1}^{\infty} \lambda^{r-1} Q^r = (1-\lambda)Q[I - \lambda Q]^{-1},$$

converges since $0 < \lambda < 1$. To complete the proof we define

$$\Delta_\lambda = I - \alpha X^T XD(I - (1-\lambda)Q[I - \lambda Q]^{-1}),$$

and show that there exists $\varepsilon > 0$ such that for any $0 < \alpha < \varepsilon$,

$$\lim_{n\to\infty} \Delta_{\lambda}^{(n)} = 0. \qquad (\diamond)$$

In this case, $[X^T \overline{w}_r^{(n)} - \overline{e}] \to 0$ as $n \to \infty$, meaning that all the estimates will tend to be correct. The proof of $(\diamond)$ can be found in the Appendix [Day92].