Off-Policy Prediction Learning: An Empirical Study of Online Algorithms

Sina Ghiassian, Banafsheh Rafiee, and Richard S. Sutton

***Abstract*— Off-policy prediction—learning the value function for one policy from data generated while following another policy—is one of the most challenging problems in reinforcement learning. This article makes two main contributions: 1) it empirically studies 11 off-policy prediction learning algorithms with emphasis on their sensitivity to parameters, learning speed, and asymptotic error and 2) based on the empirical results, it proposes two step-size adaptation methods called Step-size Ratchet and Soft Step-size Ratchet that help the algorithm with the lowest error from the experimental study learn faster. Many off-policy prediction learning algorithms have been proposed in the past decade, but it remains unclear which algorithms learn faster than others. In this article, we empirically compare 11 off-policy prediction learning algorithms with linear function approximation on three small tasks: the Collision task, the Rooms task, and the High Variance Rooms task. The Collision task is a small off-policy problem analogous to that of an autonomous car trying to predict whether it will collide with an obstacle. The Rooms and High Variance Rooms tasks are designed such that learning fast in them is challenging. In the Rooms task, the product of importance sampling ratios can be as large as 2^{14}. To control the high variance caused by the product of the importance sampling ratios, step size should be set small, which, in turn, slows down learning. The High Variance Rooms task is more extreme in that the product of the ratios can become as large as $2^{14} \times 25$. The algorithms considered are Off-policy TD(λ), five Gradient-TD algorithms, two Emphatic-TD algorithms, Vtrace, and variants of Tree Backup and ABQ that are applicable to the prediction setting. We found that the algorithms' performance is highly affected by the variance induced by the importance sampling ratios. Tree Backup(λ), Vtrace(λ), and ABTD(ζ) are not affected by the high variance as much as other algorithms, but they restrict the effective bootstrapping parameter in a way that is too limiting for tasks where high variance is not present. We observed that Emphatic TD(λ) tends to have lower asymptotic error than other algorithms but might learn more slowly in some cases. Based on the empirical results, we propose two step-size adaptation algorithms, which we collectively refer to as the Ratchet algorithms, with the same underlying idea: keep the step-size parameter as large as possible and ratchet it down only when necessary to avoid overshoot. We show that the Ratchet algorithms are effective by comparing them with other popular step-size adaptation algorithms, such as the Adam optimizer.**

Manuscript received 24 February 2022; revised 6 June 2023 and 11 November 2023; accepted 25 February 2024. This work was supported in part by DeepMind, in part by Alberta Machine Intelligence Institute, in part by the Natural Sciences and Engineering Research Council of Canada, and in part by Canadian Institute for Advanced Research. *(Corresponding author: Sina Ghiassian.)*

Sina Ghiassian was with the Department of Computing Science, University of Alberta, Edmonton, AB T6G 2R3, Canada. He is now with Spotify, Toronto, ON M5H 1W7, Canada (e-mail: ghiassia@ualberta.ca).

Banafsheh Rafiee is with the Reinforcement Learning and Artificial Intelligence (RLAI) Laboratory, University of Alberta, Edmonton, AB T6G 2R3, Canada (e-mail: rafiee@ualberta.ca).

Richard S. Sutton is with the Department of Computing Science, University of Alberta, Edmonton, AB T6G 2R3, Canada (e-mail: rsutton@ualberta.ca).

This article has supplementary downloadable material available at https://doi.org/10.1109/TNNLS.2024.3373749, provided by the authors.

Digital Object Identifier 10.1109/TNNLS.2024.3373749

***Index Terms*— Empirical study, off-policy learning, online learning, reinforcement learning.**

I. Introduction

In reinforcement learning, it is not uncommon to learn the value function for one policy while following another policy. For example, Q-learning learns the value of the greedy policy, while the agent selects its actions according to a different, more exploratory policy [1], [2]. The policy whose value function is being learned is called the *target policy*, while the more exploratory policy generating the data is called the *behavior policy*. When the two policies are different, as they are in Q-learning, the problem is said to be one of *off-policy learning*, whereas if they are the same, the problem is said to be one of *on-policy learning*. Off-policy learning is more difficult than on-policy learning and subsumes it as a special case.

There are various reasons for interest in off-policy learning. One reason is that it has been the core of many of the great successes that have come out of deep reinforcement learning in the past few years. The deep Q-networks (DQN) architecture [3] and its successors such as Double DQN [4] and Rainbow [5] rely on off-policy learning. Recent research used some modern off-policy algorithms such as Vtrace and Emphatic-TD within deep reinforcement learning (RL) architectures [6], [7].

Another reason for interest in off-policy learning is that it provides a clear way of intermixing exploration and exploitation. The dilemma is that an agent should always *exploit* what it has learned so far, but it should also always *explore* to find actions that might be superior. No agent can simultaneously behave in both ways. However, an off-policy algorithm can, in a sense, pursue both goals at the same time. The behavior policy can explore freely, while the target policy can converge to the fully exploitative, optimal policy independent of the behavior policy's explorations.

Another appealing aspect of off-policy learning is enabling learning about many policies in parallel. Once the target policy is freed from behavior, there is no reason to have a single target policy. Parallel off-policy learning of value functions has even been proposed as a way of learning general, policy-dependent, world knowledge [8], [9], [10]. Finally, numerous ideas in machine learning rely on off-policy learning, including learning temporally abstract world models [11], predictive representations of state [12], [13], auxiliary tasks [14], lifelong learning [9], and learning from historical data [15]. Other than these, off-policy learning has had many use cases that go beyond the classic online prediction case, such as in control systems and optimization [16], [17], [18].

Many off-policy learning algorithms have been explored in the history of reinforcement learning. Q-learning is perhaps the oldest [1], [2]. In the 1990s, it was realized that combining off-policy learning, function approximation, and

temporal-difference (TD) learning risked instability [19]. Off-policy algorithms with importance sampling and eligibility traces, as well as tree backup algorithms, were introduced, but they did not include a practical solution to the risk of instability [20]. Gradient-TD methods (see [21], [22]) assured stability by following the gradient of an objective function, as suggested by Baird [23]. Emphatic-TD methods [24] reweighted updates in such a way as to regain the convergence assurances of the original on-policy TD algorithms. These convergent algorithms were later developed further to learn faster resulting in algorithms such as Proximal GTD2(λ) [25], TDRC(λ) [26], and Emphatic TD(λ, β) [27]. These methods had convergence guarantees but no assurances for efficiency in practice. Other algorithms, such as Retrace [28], Vtrace [6], and ABQ [29], were developed recently to overcome difficulties encountered in practice.

As more off-policy algorithms were developed, there was a need to compare them systematically. Unfortunately, due to the computational burden, it is impossible to conduct a large comparative study in a complex environment such as the arcade learning environment (ALE). In a DQN-like architecture, many elements work together to solve a task. Each element has one or more parameters that need tuning. On the one hand, not all these parameters can be tuned systematically due to the computational cost, and on the other hand, tuning parameters carefully and studying performance over many parameters are necessary for a fair comparative study. In the original DQN work, for example, the parameters were not systematically tuned. Moreover, the original DQN agent [3] was trained for one run. A detailed comparative study, however, needs at least 30 runs and typically includes a dozen algorithms, each of which has its own parameters. For example, to compare ten algorithms on the ALE, each with 100 parameter settings, for 30 runs, we need 30 000 times more compute than what was used to train the DQN agent on an Atari game. One might think that given the increase in available compute since 2015, such a study might be feasible. Moore's law states that the available compute approximately doubles every two years. This means that compared to 2015, eight times more computing is at hand today. Taking this into account, we still need $30\,000/8 = 3750$ times more compute than what was used to train one DQN agent. This is simply not feasible now or in the foreseeable future.

Let us now examine the possibility of conducting a comparative study in a state-of-the-art domain, similar to Atari, but smaller. MinAtar [30] simplifies the ALE environment considerably but presents many of the same challenges. To evaluate the possibility of conducting a comparative study in MinAtar, we compared the training time of two agents. One agent used the original DQN architecture [3], and another used the much smaller neural network (NN) architecture used for training in MinAtar [30]. Both agents were trained for 30 000 frames on an Intel Xeon Gold 6148, 2.4-GHz CPU core. On average, each MinAtar training frame took 0.003 s, and each ALE training frame took 0.043 s. To speed up training, we repeated the same procedure on an NVidia V100SXM2 (16-GB memory) graphics processing unit (GPU). Each MinAtar training frame took 0.0023 s, and each ALE training frame took 0.0032 s. The GPU sped up the process that used a large NN (in the ALE) but did not provide much of a benefit on the smaller NN used in MinAtar. This means that, assuming that we have enough GPUs to train on, using MinAtar and ALE will not be that different. Given these data, detailed comparative studies in an environment such as MinAtar are still far out of reach.

The most meaningful empirical comparisons have been in small domains. Geist and Scherrer [31] presented the first study that compared off-policy prediction learning algorithms. Their results were complemented by Dann et al. [32] with an extra algorithm and new problems. Both studies included quadratic and linear computation algorithms. White and White [33] followed with a study on prediction learning algorithms but narrowed down the space of algorithms to the ones with linear computation, which, in turn, allowed them to go into greater detail in terms of sensitivity to parameters.

In this article, we conduct a comparative study of off-policy learning algorithms similar to previous studies. (The code for the experiments is made available at https://github.com/sinaghiassian/OffpolicyAlgorithms.) We reduce the amount of required computation in this study in three ways. First, we focus on comparing off-policy algorithms and remove other confounding factors from the comparison. The comparison will not include elements such as complex optimizers, target networks, or experience replay buffers. Second, we focus on linearly learning the value function from fixed given features. While nonlinear learning systems are of special interest, sometimes, it is fruitful when performing careful comparisons between algorithms to focus on the linear case and simple problems, as, for example, was done in the original work on TD learning, recursive least squares, and eligibility traces. This is often appropriate when the issue involved is a basic one that does not strongly interact with the nonlinear aspect, which we believe is the case for our study of off-policy learning. Focusing on linearly learning the value function through fixed features is also justified through the two-timescale view of NNs [34]. In this view, it is assumed that the features are learned using the first $n-1$ layers of the NN at their own timescale, and then, the features are used by the last layer to linearly learn the value function. Third, we focus on fully incremental online algorithms. Many algorithms referred to as the off-policy evaluation (OPE) family of algorithms assume access to data beyond what the agent experiences at each time step. This article focuses on the fully incremental setting, in which the agent makes one interaction with the environment, receives a reward, learns from it, and then discards the sample and moves to the next step. This is in contrast to the setting in which the agent has access to historical data. Not having access to historical data, the agent is more limited in what it can learn.

This article is similar to previous studies in that it treats prediction with linear function approximation and similar to the study by White and White [33] in restricting attention to linear complexity algorithms. Our study differs from earlier studies in that it treats more algorithms and does a deeper empirical analysis. The additional algorithms are the prediction variants of Tree Backup(λ) [35], Retrace(λ) [28], ABQ(ζ) [29], and TDRC(λ) [26]. Our empirical analysis is deeper primarily in that we examine the dependence of all 11 algorithms' performance on all of their parameters individually. Our results, though limited to relatively small tasks, are a significant addition to what is known about the comparative performance of off-policy learning algorithms.

The focus of our study is on one of the most important challenges of off-policy learning: the problem of slow learning. Slow learning and high variance are sometimes used interchangeably in off-policy learning. The reason is that when

large importance sampling ratios are used, small step-size parameters are needed to control the high variance induced by these ratios, which, in turn, results in slow learning. We consider three tasks with the product of importance sampling ratios increasing from each task to the next. We explore the whole parameter space of algorithms and conclude that the problem variance heavily affects the algorithm performance. We propose two step-size adaptation algorithms that help the algorithm with the lowest error level learn faster.

II. Formal Framework

We simulate the agent-environment interaction using the MDP framework. An agent and an environment interact at discrete time steps, $t = 0, 1, \ldots$ At each time step the environment is in a state $S_t \in \mathcal{S}$ and chooses an action, $A_t \in \mathcal{A}$ under a behavior policy b: $\mathcal{A} \times \mathcal{S} \to [0, 1]$. For a state and an action (s, a), the probability that action a is taken in state s is denoted by $b(a|s)$ where "|" means that the probability distribution is over a for each s. After choosing an action, the agent receives a numerical reward $R_{t+1} \in \mathcal{R} \subset \mathbb{R}$, and the environment moves to the next state S_{t+1}. The transition from S_t to S_{t+1} depends on the MDP's transition dynamics.

In off-policy learning, the policy the agent learns about is different from the policy the agent uses for behavior. The policy that the agent learns about is denoted by π and is termed the *target policy*, whereas the policy that is used for behavior is denoted by b and is termed the *behavior policy*. The goal is to learn the expectation of the sum of the future rewards, the *return*, under a target policy. Both target and behavior policies are fixed in prediction learning. The return includes a termination function, which can be also thought of as a generalized notion of discounting: γ: $\mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$

$$G_t \stackrel{\text{def}}{=} R_{t+1} + \gamma(S_t, A_t, S_{t+1})R_{t+2} + \gamma(S_t, A_t, S_{t+1})\gamma(S_{t+1}, A_{t+1}, S_{t+2})R_{t+3} + \cdots.$$

If, for some triplet, S_k, A_k, and S_{k+1}, the termination function returns zero, the accumulation of the rewards is terminated.

The expectation of the return when starting from a specific state and following a specific policy thereafter is called the *value* of the state under the policy. The value function under policy π is defined as $v_\pi(s) \stackrel{\text{def}}{=} \mathbb{E}_\pi[G_t|S_t = s]$, where $\mathbb{E}_\pi[G_t|S_t]$ is the conditional expectation of the return given that the agent starts in $S_t = s$ and follows π thereafter. Because the agent is following b not π, we will need to correct for the difference between the behavior and target policies. These corrections are often done using importance sampling ratios

$$\rho_t \stackrel{\text{def}}{=} \frac{\pi(A_t|S_t)}{b(A_t|S_t)}. \tag{1}$$

At time steps where the target and behavior policies have the same distribution and in on-policy learning, the ratio is one.

For any random variable X_{t+1} that is generated using the behavior policy and depends on the state–action–next state triplet, the expectation under the target policy can be computed using importance sampling ratios

$$\mathbb{E}_b\big[\rho_t X_{t+1}\big|S_t = s\big] = \sum_a b(a|s)\frac{\pi(a|s)}{b(a|s)}X_{t+1} = \mathbb{E}_\pi\big[X_{t+1}\big|S_t = s\big] \quad \forall s \in \mathcal{S}.$$

In many problems of interest, the state space is large and the value function should be approximated using limited resources. We use linear functions to approximate $v_\pi(s)$. The approximate value function, $\hat{v}(\cdot, \mathbf{w})$, is a linear function of a weight vector $\mathbf{w} \in \mathbb{R}^d$. Corresponding to each state s, there is a d-dimensional vector, $\mathbf{x}(s)$, where $d \ll |\mathcal{S}|$. The approximate value for a state is $\hat{v}(s, \mathbf{w}) \stackrel{\text{def}}{=} \mathbf{w}^\top \mathbf{x}(s) \approx v_\pi(s)$.

III. Algorithms

We briefly introduce the 11 algorithms used in our empirical study. These 11 are intended to include all the best candidates for off-policy prediction learning with linear function approximation. The complete update rules of all algorithms and additional technical discussion can be found in Appendices A and B, respectively (see the Supplementary Material).

Off-policy TD(λ) [20] is the off-policy variant of the original TD(λ) algorithm [36] that uses importance sampling to reweight the returns and account for the differences between the behavior and target policies. This algorithm has just one set of weights and one step-size parameter.

We include five algorithms from the Gradient-TD family. *GTD(λ)* and *GTD2(λ)* are based on algorithmic ideas introduced by Sutton et al. [22] and then extended to eligibility traces [21]. *Proximal GTD2(λ)* [25], [37] is a "mirror descent" version of GTD2 using a saddle point objective function. These algorithms approximate stochastic gradient descent (SGD) on an alternative objective function, the mean squared projected Bellman error (MSPBE). *HTD(λ)* [33], [38] is a "hybrid" of GTD(λ) and TD(λ), which becomes equivalent to classic TD(λ) where the behavior policy coincides with the target policy. *TDRC(λ)* [26] is a recent variant of GTD(λ) that adds regularization. All these methods involve an additional set of learned weights (beyond that used in $\hat{v}$) and a second step-size parameter, which can complicate their use in practice. TDRC(λ) offers a standard way of setting the second step-size parameter, which makes this less of an issue. All of these methods are guaranteed to converge with an appropriate setting of their two step-size parameters.

We include two algorithms from the Emphatic-TD family. Emphatic-TD algorithms attain stability by upweighting or downweighting the updates made on each time step by Off-policy TD(λ). If this variation in the emphasis of updates is done in just the right way, stability can be guaranteed with a single set of weights and step size. *Emphatic TD(λ)* was introduced by Sutton et al. [24]. The variant *Emphatic TD(λ, β)* [27] has an additional parameter, $\beta \in [0, 1]$, intended to reduce variance.

The final three algorithms in our study—ABTD(ζ), Vtrace(λ), and the prediction variant of Tree Backup(λ)—can be viewed as attempts to address the problem of large variations in the product of importance sampling ratios. If this product becomes large, the step-size parameter must be set small to prevent overshoot, and then, learning may be slow. All these methods attempt to control the importance of sampling products by changing the bootstrapping parameter from step to step [39]. Munos et al. [28] proposed simply putting a cap on the importance sampling ratio at each time step; they explored the theory and practical consequences of this modification in a control context with their Retrace algorithm. *Vtrace(λ)* [6] is a modification of Retrace to make it suitable for prediction rather than control. Mahmood et al. [29] developed a more flexible algorithm that achieves a similar effect. Their algorithm was also developed for control; to apply the idea to prediction learning, we had to develop a nominally new algorithm, *ABTD(ζ)*, which naturally extends ABQ(ζ) from control to prediction. [ABTD(ζ) is developed in Appendix D (see the Supplementary Material).] Finally, *Tree Backup(λ)* [35] reduces the effective λ by the probability of the action

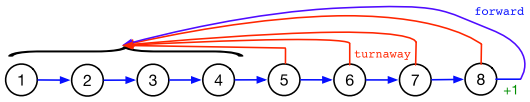


Fig. 1. Collision task. When the `forward` action is taken in the last state, a collision is said to occur and a reward of 1 is received.

taken at each time step. Each of these algorithms has been shown to be effective on specific problems.

IV. Collision Task

The Collision task is an idealized off-policy prediction-learning task. A vehicle moves along an eight-state track toward an obstacle with which it will collide if it keeps moving forward. In the first four states, `forward` is the only possible action, whereas, in the last four states, two actions are possible: `forward` and `turnaway` (see Fig. 1). The `forward` action always moves the vehicle one state further along the track; if it is taken in the last state, then a collision is said to occur, the reward becomes 1, and the vehicle transitions into one of the first four states at random equiprobably. The `turnaway` action causes the vehicle to "turn away" from the wall, which also causes the vehicle to transition into one of the first four states with equal probability, except with a reward of zero. The reward is also zero on all other transitions. The termination function returns 0.9 at all transitions except for when transitioning from the last state with the `forward` action and transitioning from any state when the `turnaway` action is taken. The task is continuing. The target policy is to always take the `forward` action, $\pi(\texttt{forward}|s) = 1, \forall s \in \mathcal{S}$. The behavior policy is to take the two actions with equal probability, $b(\texttt{forward}|s) = b(\texttt{turnaway}|s) = 0.5, \forall s \in \{5, 6, 7, 8\}$. We are seeking to learn the value function for the target policy, which, in this case, is $v_\pi(s) = \gamma^{8-s}$.

This idealized task is roughly analogous to and involves some similar issues as real-world autonomous driving problems, such as exiting a parallel parking spot without hitting the car in front of you, or learning how close you can get to other cars without risking collisions. In particular, if these problems can be treated as off-policy learning problems, then solutions can potentially be learned with fewer collisions. In this article, we are testing the efficiency of various off-policy prediction-learning algorithms to maximize how much they learn from the same number of collisions.

Similar problems have been studied using mobile robots. White [9] and Rafiee et al. [40] used off-policy learning algorithms running on an iRobot Create to predict collisions as signaled by activation of the robot's front bumper sensor. Modayil and Sutton [41] trained a custom robot to predict motor stalls and turn off the motor when a stall was predicted.

We artificially introduce function approximation into the Collision task. Although a tabular approach is entirely feasible on this small problem, it would not be on the large problems of interest. In real applications, the agent would have sensor readings, which will go through an artificial NN to create feature representations. We simulate such representations by randomly assigning to each state a binary feature vector $\mathbf{x}(s) \in \{0, 1\}^d, \forall s \in \{1, \ldots, 8\}$. We chose $d = 6$ so that was not possible for all eight of the feature vectors (one per state) to be linearly independent. In particular, we chose all eight feature vectors to have exactly three 1s and three 0s, with the location of the 1s for each state being chosen randomly.

Because the feature vectors are linearly dependent, it is not possible in general for a linear approximation, $\hat{v}(s, \mathbf{w}) = \mathbf{w}^\top \mathbf{x}$, to equal to $v_\pi(s)$ at all eight states of the Collision task. (See Appendix F (Supplementary Material) for a comparison between $v_\pi(s)$ and its linear approximations with linearly dependent features.)

Given a feature representation $\mathbf{x} : \mathcal{S} \to \mathbb{R}^d$, a linear approximate value function is determined by its weight vector $\mathbf{w} \in \mathbb{R}^d$, the quality of which is assessed by its squared error at each state weighted by how often each state occurs

$$\overline{\text{VE}}(\mathbf{w}) = \sum_{s \in \mathcal{S}} \mu_b(s) \big[\hat{v}(s, \mathbf{w}) - v_\pi(s)\big]^2 \tag{2}$$

where $\mu_b(s)$ is the state distribution approximated from visitation counts from one million sample time steps of the behavior policy.

V. Collision Task Experimental Setup and Results

The Collision task, in conjunction with its behavior policy, was used to generate 20 000 time steps, comprising one *run*, and then, this was repeated for a total of 50 independent runs.

Each run also used a different feature representation randomly generated as described in Section IV. Focusing on one-hot representations, we decided to choose a different random representation for each of the 50 runs to study the performance of algorithms across various representations. The 11 algorithms were then applied to the 50 runs, each with a range of parameter values; each combination of algorithm and parameter settings is termed an *algorithm instance*. (See Appendix F (Supplementary Material) for a sample learning curves.)

A list of all parameter settings used can be found in Appendix G (Supplementary Material). They included 12 values of λ, 19 values of α, 15 values of η (for the Gradient-TD family), six values of β, and 19 values of ζ, for approximately 20 000 algorithm instances in total. In each run, the weight vector was initialized to $\mathbf{w}_0 = \mathbf{0}$ and then updated at each step by the algorithm instance to produce a sequence of $\mathbf{w}_t$. At each step, we also computed and recorded $\overline{\text{VE}}(\mathbf{w}_t)$.

A. Main Results

As an overall measure of the performance of an algorithm instance, we take its learning curve over 50 runs and then average it across the 20 000 steps. In this way, we reduce all the data for an algorithm instance to a single number that summarizes performance. These numbers appear as points in our main results figure (see Fig. 2). Each figure of the figure is devoted to a single algorithm.

For example, performance numbers for instances of Off-policy TD(λ) are shown as points in the left figure of the second row of Fig. 2. This algorithm has two parameters: the step-size parameter, α, and the bootstrapping parameter, λ. The points are plotted as a function of α, and the points with the same λ value are connected by lines. The blue line shows the performances of the instances of Off-policy TD(λ) with $\lambda = 1$, the red line shows the performances with $\lambda = 0$, and the gray lines show the performances with intermediate λs. Note that all the lines are U-shaped functions of α, as is to be expected; at small α, learning is too slow to make much progress; and at large α, there is overshoot and divergence, as in the blue line in Fig. 19 in Supplementary Material. For each point, the standard error over the 50 runs is also given as an error bar though these are too small to be seen in all except the rightmost points of each line where the step size was highest and divergence was common.

The first focus on the blue line (of the left figure on the second row of Fig. 2), representing the performances of Off-policy TD(λ) with $\lambda = 1$. There is a wide sweet spot,

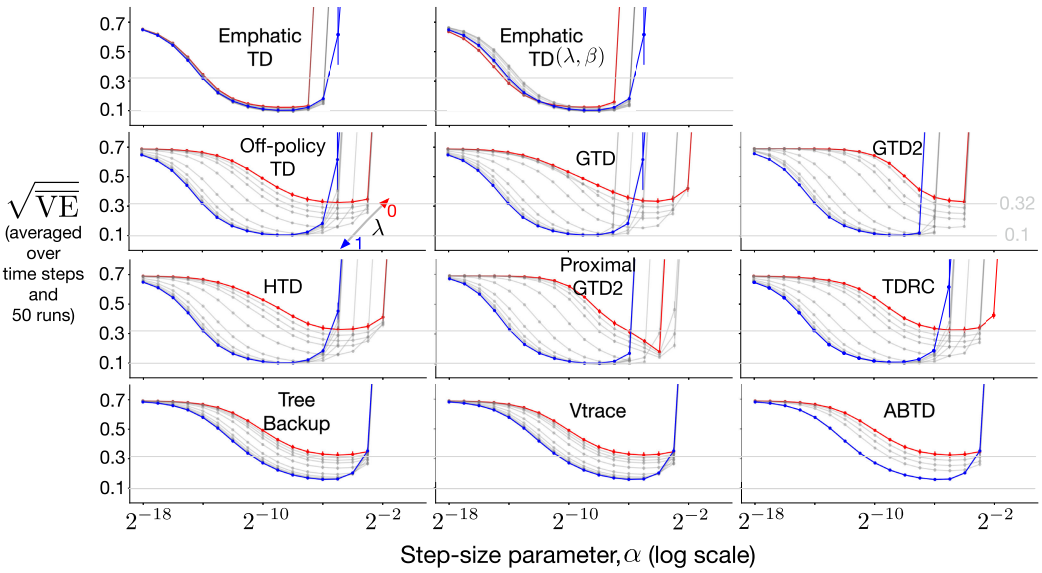


Fig. 2. Performance of all algorithms on the Collision task as a function of their parameters α and λ. The red curves show the performance with $\lambda = 0$; the blue curves show the performance with $\lambda = 1$; and the gray curves show the performance with intermediate values of λ. The top tier algorithms (top row) attained a low error ($\approx$0.1) at all λ values. The middle tier of six algorithms attained a low error for $\lambda = 1$, but not for $\lambda = 0$. The bottom tier of three algorithms was unable to reach an error of $\approx$0.1 at any λ value.

that is, there are many intermediate values of α at which good performance (low average error) is achieved. Note that the step-size parameter α is varied over a wide range, with logarithmic steps. The minimal error level of about 0.1 was achieved over four or five powers of two for α. This is the primary measure of good performance that we look for: low error over a wide range of parameter values.

Now, contrast the blue line with the red and gray lines for Off-policy TD(λ) (see Fig. 2, the second row left figure). Recall that the blue line is for $\lambda = 1$, the red line is for $\lambda = 0$, and the gray lines are for intermediate values of λ. First note that the red line shows generally worse performance; the error level at $\lambda = 0$ was higher, and its range of good α values was slightly smaller (on a logarithmic scale). The intermediate values of λ all had performances that were between the two extremes. Second, the sweet spot consistently shifted right, toward higher α, as λ was decreased from 1 to 0.

Now, armed with a thorough understanding of the Off-policy TD(λ) figure, consider the other figures of Fig. 2. Overall, there are a lot of similarities between the algorithms and how their performances varied with α and λ. For all algorithms, error was lower for $\lambda = 1$ (the blue line) than for $\lambda = 0$ (the red line). Bootstrapping apparently confers no advantage in the Collision task for any algorithm.

The most obvious difference between algorithms is that the performance of the two Emphatic-TD algorithms varied relatively little as a function of λ; their blue and red lines are almost on top of one another, whereas those of all the other algorithms are qualitatively different. The Emphatic algorithms generally performed as well as or better than the other algorithms. At $\lambda = 1$, the Emphatic algorithms reached the minimal error level of all algorithms ($\approx$0.1), and their ranges of good α values were as wide as that of the other algorithms, while at $\lambda = 0$, the best errors of the Emphatic algorithms were qualitatively better than those of the other algorithms. The minimal $\lambda = 0$ error level of the Emphatic algorithms was about 0.15 compared to approximately 0.32 (shown as a second thin gray line) for all the other algorithms (except Proximal GTD2, a special case that we consider later). Moreover, for the Emphatic algorithms, the sweet spot for α shifted a little as λ varied. The shift was markedly less than for the six algorithms in the middle two rows of Fig. 2. The lack of an interaction between the two parameter values is another potential advantage of the Emphatic algorithms.

The lowest error level for 8 of the algorithms was $\approx$0.1 (shown as a thin gray line), and for the other three algorithms, the best error was higher, $\approx$0.16. The differences between 8 and 3 were highly statistically significant, whereas the differences within the two groups were negligible. The three algorithms that performed worse than the others were Tree Backup(λ), Vtrace(λ), and ABTD(ζ)—shown in the bottom row of Fig. 2. The difference was only for large λs; at $\lambda = 0$, these three algorithms reached the same error level ($\approx$0.32) as the other non-Emphatic algorithms. The three worse algorithms' range of good α values was also slightly smaller than the other algorithms (with the partial exception, again, of Proximal GTD2(λ)). A mild strength of the three is that the best α value shifted less as a function of λ than for the other six non-Emphatic algorithms. Generally, the performances of these three algorithms in Fig. 2 look very similar as a function of parameters. An interesting difference is that for ABTD(ζ), we only see three gray curves, whereas for the other two algorithms, we see seven. For ABTD(ζ), there is no λ parameter, but the parameter ζ plays the same role. In our experiment, ABTD(ζ) performed identically for all ζ values greater than 0.5; four gray lines with different ζ values are hidden behind ABTD's blue curve.

In summary, our main result is that on the Collision task, the performances of the 11 algorithms fell into three tiers. In the top tier are the two Emphatic-TD algorithms, which performed well and almost identically at all values of λ and significantly better than the other algorithms at low λ. Although this difference did not affect the best performance here (where $\lambda = 1$ is best), the ability to perform well with bootstrapping is expected to be important on other tasks. In the middle tier are Off-policy TD(λ) and all the Gradient-TD algorithms, all of which performed well at $\lambda = 1$ but less well at $\lambda = 0$. Finally, in the bottom tier are Tree Backup(λ), Vtrace(λ), and ABTD(λ), which performed very similarly and not, as well

as the other algorithms at their best parameter values. All of these differences are statistically significant, albeit specific to this one task. In Fig. 2, the three tiers are the top row, the two middle rows, and the bottom row.

The reason why Emphatic TD algorithms reached a lower error level than some others might be the objective function that they minimize. Emphatic TD algorithms minimize the Emphatic weighted MSPBE. This is in contrast to all other algorithms studied in this article, which minimize the behavior policy-weighted MSPBE. In our results, the error measure is the mean squared value error ($\overline{\text{VE}}$): the difference between the true value function and the value function found by an algorithm. Our results suggest that the distance between the minimums of Emphatic weighted MSPBE and $\overline{\text{VE}}$ is smaller than the distance between the minimums of behavior-weighted MSPBE and $\overline{\text{VE}}$.

The reason why ABTD, Tree Backup, and Vtrace did not perform and others is probably that they cut off the importance sampling ratio, which in turn introduces bias into the solution and causes the algorithm to converge to a higher error level. On the other hand, one can expect that these algorithms perform better than others on problems where importance sampling ratios are really large.

B. Emphatic-TD

So far, we looked at performance as a function of α and λ. We now set $\lambda = 0$ and study the effect of β on the performance of Emphatic TD(λ, β). We focus on the full bootstrapping case ($\lambda = 0$) because the largest differences were observed in this case. The curves shown in Fig. 2 are for the best values of β, meaning that, for each λ, we found the combination of α and β that resulted in the minimum average error, fixed β, and plotted the sensitivity for that fixed β over α. Here, we show how varying β affects performance.

The errors of Emphatic TD(0) and Emphatic TD(0, β) for various values of α and β are shown in Fig. 3. Both algorithms performed similarly well on the Collision task, meaning that they both had a wide sensitivity curve and reached the same ($\approx$0.1) error level. Notice that, as β increased, the sensitivity curve for Emphatic TD(0, β) shifted to the left and the error overall decreased. With $\beta = 0$, Emphatic TD(λ, β) reduces to TD(λ). With $\beta = 0.8$ and $\beta = 1$, Emphatic TD(λ, β) reached the same error level as Emphatic TD(λ). With $\beta = \gamma$, Emphatic TD(λ, β) reduces to Emphatic TD(λ). This explains why the red curve is between the $\beta = 0.8$ and $\beta = 1$ curves.

The results make it clear that the superior performance of Emphatic methods is almost entirely due to the basic idea of emphasis; the additional flexibility provided by β of the Emphatic TD(λ, β) was not important on the Collision problem.

C. Gradient-TD

To study Gradient-TD algorithms in more detail, we set $\lambda = 0$ and analyze the effect of η on performance, where $\alpha = \eta * \alpha_{\mathbf{v}}$ and $\alpha_{\mathbf{v}}$ is the second step size. Previously, in Fig. 2, we looked at the results with the best values of η for each λ; meaning that, for each λ, first, the combination of α and η that resulted in the lowest average $\overline{\text{VE}}$ was found, and then, sensitivity to step size was plotted for that specific value of η. Sensitivity to step size for various values of η with $\lambda = 0$ is shown in Fig. 4. Each figure shows the result of two Gradient-TD algorithms for various ηs: one main algorithm, shown with solid lines, and another additional algorithm shown with dashed lines for comparison. The first focuses on the upper left figure. The

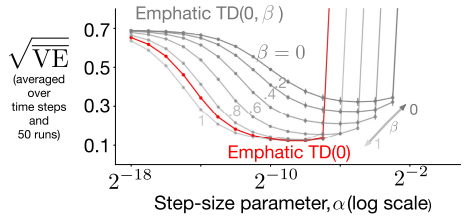


Fig. 3. Detail on the performance of Emphatic TD(λ, β) at $\lambda = 0$. Note that Emphatic TD(λ) is equivalent to Emphatic TD(λ, γ), and, here, $\gamma = 0.9$. The flexibility provided by β does not help on the Collision task.

upper left figure shows the parameter sensitivity for GTD2(0) for four values of η, and in addition, it shows GTD(0) results as dashed lines for comparison (for results with more values of η, see Appendix H (Supplementary Material)). The color for each value of η is consistent within and across the four figures, meaning that, for example, $\eta = 256$ is shown in green in all figures, either as dashed or solid lines. For all parameter combinations, GTD errors were lower than (or similar to) GTD2 errors. With two smaller values of η (1 and 0.0625), GTD had a wider and lower sensitivity curve than GTD2, which means that GTD was easier to use than GTD2.

Let us now move on to the upper right figure of Fig. 4. Proximal GTD2 had the most distinctive behavior among all Gradient-TD algorithms. With $\lambda = 0$, in some cases, it converged to a lower error than all other Gradient-TD algorithms. Proximal GTD2 was more sensitive to the choice of α than other Gradient-TD algorithms except for GTD2. Proximal GTD2 had a lower error and a wider sensitivity curve than GTD2. To see this, compare the dotted and solid lines in the upper right figure of Fig. 4.

In the lower left figure, we see that GTD and HTD performed similarly. Sensitivity curves were similarly wide but HTD reached a lower error in some cases. We see this by comparing the dotted and solid pink curves in the lower left figure.

The fourth figure shows sensitivity to the step-size parameter for HTD and TDRC. Notice that TDRC has one sensitivity curve, shown in dashed blue. This is because η is set to one (also its regularization parameter was set to one) as proposed in the original paper. HTD's widest curve was with $\eta = 0.0625$, which was as wide as TDRC's curve. For a more in-depth study of TDRC's extra parameters, see Appendix H (Supplementary Material).

On the one hand, among the Gradient-TD algorithms, TDRC was the easiest to use. On the other hand, in the case of full bootstrapping, Proximal GTD2 reached the lowest error level. The fact that Proximal GTD2 converged to a lower error level might be due to a few different reasons. One possible reason is that it might not have converged to the minimum of the MSPBE, such as other Gradient-TD algorithms. Another reason might be that it converged to a minimum of the projected Bellman error that was different from the minimum the other algorithms converged to. Further analysis is required to investigate this.

VI. Rooms Task

The Rooms task uses a variant of the four-room environment MDP [11]: a gridworld with 104 states, roughly partitioned into four contiguous areas called rooms (see Fig. 5). These rooms are connected through four hallway states. Four deterministic actions are available in each state: `left`, `right`, `up`, and `down`. Taking each action results in moving in the corresponding direction except for cells neighboring a wall in

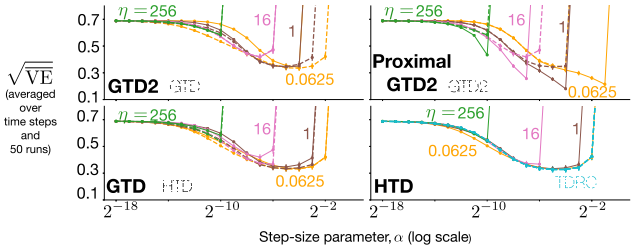


Fig. 4. Detail on the performance of Gradient-TD algorithms at $\lambda = 0$. Each algorithm has a second step-size parameter, scaled by η. A second algorithm's performance is also shown in each figure, with dashed lines, for comparison.

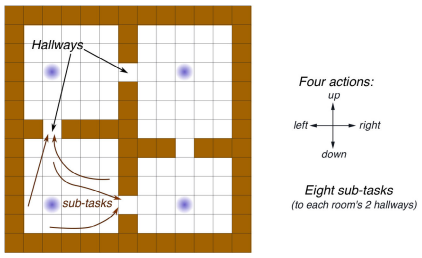


Fig. 5. Four-room environment. Four actions are possible in each state. Two hallway states are shown with arrows. Four subtasks are also schematically shown. The four shaded states are the ones where the Rooms and High Variance Rooms tasks have different policies.

which the agent will not move if it takes the action toward the wall. The task is continuing.

The Rooms task consists of eight subtasks. Solving a subtask corresponds to learning the value function for the target policy corresponding to the subtask. Under the corresponding target policy, the agent follows the shortest path to one of the room's hallways, which we refer to as the corresponding hallway. If two actions are optimal in a state, one of the two is taken randomly with equal probability. The termination function returns 0.9, while the agent is in the room. Once the agent reaches the corresponding hallway, the termination function returns 0, without affecting the actual trajectory of the behavior policy. The termination function remains 0 for all states that are not part of the subtask. When the agent reaches the corresponding hallway, it receives a reward of 1. The reward for all other transitions is 0. In the same room, another target policy is defined under which the agent follows the shortest path to the other hallway. Therefore, there are exactly two subtasks defined for each state, including the hallways.

The Rooms task is designed to be a high-variance problem. By a high-variance problem, we mean that the product of importance sampling ratios can vary between small and large values during learning and can cause large changes in the learned weight vector that might make learning unstable. Under the target policy, the agent follows the shortest path to a hallway, and the behavior policy is equiprobable random. If the agent is in the top left state of the upper right room, chooses the `right` action twice, and then chooses the `down` action six times, the product of the importance sampling ratios will become 2^{14} because the importance sampling ratio is 2 at the first two time steps, and it is $1/(1/4) = 4$ for a total of six steps.

The agent learns about two subtasks at each time step. For Emphatic TD, this will be automatically enforced with the interest function set to 0 for all states that are not part of the active subtask. Other algorithms do not natively have interest, so we enforce this manually by making sure that at each time step, the agent knows what subtasks are active and only updates weight vectors corresponding to those subtasks.

We used linear function approximation to solve the task. To represent states, (x, y) coordinates were tile coded. The x and y coordinates both ranged from 0 to 10, where $x = 0$ and $y = 0$ represent the far left and bottom cells, respectively. We used four tilings, each of which was 2×2 tiles. The features used to solve the task can be produced using any system, for example, an NN. Our focus, in this article, is on learning the value function linearly using known features, the task that is typically carried out by the last layer of the NN.

To assess the quality of the value function found by an algorithm, we used the mean squared value error

$$\overline{\text{VE}}(\mathbf{w}) = \frac{\sum_{s\in\mathcal{S}} \mu_b(s) i(s) \big[\hat{v}(s,\mathbf{w}) - v_\pi(s)\big]^2}{\sum_{s\in\mathcal{S}} \mu_b(s) i(s)}$$

where $i(s)$, the *interest function*, $i : \mathcal{S} \to \{0, 1\}$, defines a weighting over states and $\mu_b(s)$ is an approximation of the stationary distribution under the behavior policy, which was calculated by having the agent start at the bottom left corner and follow the behavior policy for a hundred million time steps and computing the fraction of time the agent spent in each state. The true value function was calculated by following each of the target policies from each state to their corresponding hallway once. The interest function is one for all states where the target policy is defined. Setting $i(s)$ in the error computation ensures that prediction errors from states outside of a room do not contribute to the error computed for each subtask. We computed the square root of $\overline{\text{VE}}$ for each policy separately and then simply averaged the errors of the eight approximate value functions to compute an overall measure of error, which we denote by $\text{A}\sqrt{\overline{\text{VE}}}$: $\text{A}\sqrt{\overline{\text{VE}}}(\mathbf{w}) \overset{\text{def}}{=} (1/8)\sum_{j=1}^{8} (\overline{\text{VE}}(\mathbf{w}^j))^{1/2}$.

VII. Rooms Experimental Setup and Results

The task and the behavior policy were used to generate 50 000 steps, comprising one *run*. This was repeated for a total of 50 runs. The algorithm instances applied to the task were the same as the Collision task. At the beginning of each run, the weight vector was initialized to $\mathbf{w}_0 = \mathbf{0}$ and then was updated at each step by the algorithm instance. At each time step, $\text{A}\sqrt{\overline{\text{VE}}}$ was computed and recorded.

A. Main Results

The performance of an algorithm instance is summarized by $\text{A}\sqrt{\overline{\text{VE}}}$ averaged over runs and time steps and shown for many algorithm instances in Fig. 6.

The measure of good performance that we look for in the data is low error over a wide range of parameters. For Off-policy TD(λ), the bottom of the U-shaped curves was at about 0.14 (shown as a thin gray line). The instances that reached this error level were in a sweet spot. This sweet spot was large for Off-policy TD(λ).

There are lots of similarities between the algorithms. All algorithms had their best performance with intermediate values of λ, except for Tree Backup(λ), Vtrace(λ), and ABTD(ζ). All algorithms except the three reached about 0.14 error level. With all algorithms, except for Emphatic TD(λ), the sweet spot shifted to the left, as λ increased from 0 to 1. Between the five Gradient-TD algorithms, GTD2 and Proximal GTD2 were more sensitive to α, and their U-shaped curves were less smooth than some others.

One of the most distinct behaviors was observed with Emphatic TD(λ) whose performance changed little as a function of λ. Its best performance was a bit worse than 0.14 and

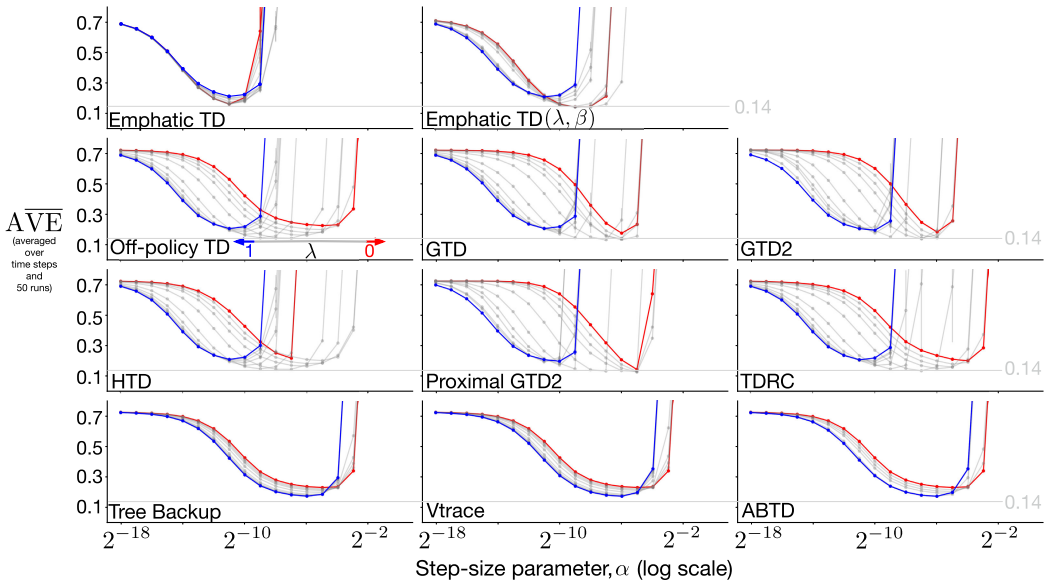


Fig. 6. Error as a function of α and λ for all algorithms on the Rooms task. All algorithms reached the 0.14 error level except Tree Backup(λ), Vtrace(λ), and ABTD(ζ). Proximal GTD2(λ) and Emphatic TD(λ) were more sensitive to α than other algorithms. Emphatic TD(λ) was less sensitive to λ than other algorithms.

was achieved with $\lambda = 0$. Emphatic TD(λ) was more sensitive to α than other algorithms. Notice how its U-shaped curve is less smooth and narrower at its bottom than many others.

Tree Backup, Vtrace, and ABTD behaved similarly. They did not reach the 0.14 error level and had their best performance at $\lambda = 1$. ABTD(ζ) was less sensitive to its bootstrapping parameter, ζ, and only had three gray curves, whereas Vtrace(λ) and Tree Backup(λ) had more gray curves. Many of the ABTD(ζ) gray curves are hidden behind the blue curve.

Overall, on the Rooms task, we divide the algorithms into two tiers. All algorithms except Tree Backup, Vtrace, and ABTD had an error close to 0.14 and are in the first tier. Tree Backup, Vtrace, and ABTD are in the second tier because, regardless of how their parameters were set, they never reached the 0.14 error level. These conclusions are in some cases similar to the ones from the Collision task. When applied to the Collision task, algorithms were divided into three tiers: Emphatic-TD algorithms were in the top tier, Gradient-TD and Off-policy TD(λ) were in the middle tier, and Tree Backup, Vtrace, and ABTD were in the bottom tier. Similar to the Collision task, Tree Backup, Vtrace, and ABTD did not perform as well as other algorithms when applied to the Rooms task. Unlike the Collision task, Emphatic TD's best performance was similar to Gradient-TD algorithms' best performance, but not better.

B. Emphatic-TD

We now set $\lambda = 0$ and study the effect of β on the performance of Emphatic TD(λ, β). Errors for various βs are plotted in the left figure of Fig. 7.

The best performance was achieved with an intermediate β and was statistically significantly lower than the error of Emphatic TD. This is in contrast to the results on the Collision task in which no improvement was observed by varying β. It seems like varying β might only be useful in cases where the problem variance is high.

Two learning curves and two dashed straight lines are shown in the right figure of Fig. 7. The learning curves correspond to algorithm instances of Emphatic TD(0) and Emphatic TD(0, β) that minimized the area under the learning curve (AUC). The dashed lines show the approximate solutions of Emphatic TD(0) and Off-policy TD(0). These solutions are found using all the data over 50 000 time steps and 50 runs and least-squares algorithms for which the update rules are provided in Appendix A (see the Supplementary Material). These solutions show the error level these algorithms would converge to if they were applied to the task with a small enough α and were run for long enough.

Emphatic TD(0) learned slower than Emphatic TD(0, β). In fact, Emphatic TD(λ) learned slower than all other algorithms. Learning curves for algorithm instances with the smallest AUC for all algorithms for general λ are shown in Fig. 8. However, if Emphatic TD had enough time to learn, it would converge to a lower asymptotic solution than other algorithms, as shown by the straight dashed lines in Fig. 7.

The results from the Collision and Rooms task collectively show that Emphatic TD(λ) tends to have a lower asymptotic error level but is more prone to problem variance. On the Collision task, Emphatic TD(λ) had a lower asymptotic error level and learned faster than other algorithms. Moving on to the Rooms task, Emphatic TD learned slower than other algorithms but still had a lower asymptotic error.

C. Gradient-TD

We now turn to studying the Gradient-TD algorithms in more detail. Errors for $\lambda = 0$ for various η are plotted in Fig. 9.

All algorithms solved the problem fairly well. According to the upper right figure, Proximal GTD2 had the lowest error among all algorithms but only with one of its parameter settings. Proximal GTD2 had a lower error than GTD2 for small η. For larger η, the reverse was true. According to the lower left figure, GTD had a slightly lower error than HTD; however, HTD was less sensitive to α, specifically with $\eta = 0.0625$. According to the lower right figure, TDRC's bowl was almost as wide as HTD's widest bowl.

Although Proximal GTD2 performed better than others, it did so with only one parameter setting, and thus, the improvement it provides is not of much practical importance. HTD, GTD, and TDRC all performed well and were robust to the choice of α. TDRC, specifically, with one tuned parameter, is the easiest-to-use algorithm for solving the Rooms task.

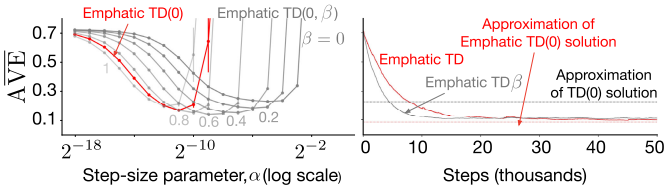


Fig. 7. Detail on the Emphatic TD(λ, β) performance on the Rooms task at $\lambda = 0$ is shown on the left. The best learning curves for each algorithm are shown on the right. The β parameter helped Emphatic TD(λ, β) learn faster.

VIII. High Variance Rooms Task

With a slight modification to the Rooms task, we increased its variance. We changed the behavior policy in four states such that one action is chosen with 0.97 and the three other actions with 0.01 probability. These states are shaded in blue in Fig. 5. In the two left rooms, the `left` action is chosen with 0.97 probability and, in the two right rooms, the `right` action. This means that if the `down` action is chosen in the blue state in the upper right room, the importance sampling will be $1/(1/100)$. The new task is called the High Variance Rooms task. If the agent starts from the upper left state in the upper right room and takes two `right` actions and then six `down` actions, the product of importance sampling ratios will be $2^{14} \times 25$. In addition to more extreme importance sampling ratios, this small change in the behavior policy largely changes the state visitation distribution compared to the Rooms task. The states to the left of the blue states in the two left rooms and the states to the right of the blue states in the two right rooms are visited more often.

IX. High Variance Rooms Experimental Setup and Results

The experimental setup for this task was the same as the Rooms task. The number of time steps, the number of runs, and the algorithm instances applied to the task were all the same.

A. Main Results

The main results for the High Variance Rooms task are plotted in Fig. 10. The variance caused by the importance sampling ratio impacted all algorithms except Tree Backup(λ), Vtrace(λ), and ABTD(ζ). These three algorithms reached an error of 0.2 (shown as a thin gray line), which was the lowest error achieved on this task. Similar to the Rooms and the Collision tasks, these three algorithms had their best performance with $\lambda = 1$.

Now, let us focus on the rest of the algorithms in the three first rows of Fig. 10. All algorithms except Emphatic TD(λ), Proximal GTD2(λ), and GTD2(λ) reached the 0.23 error level (shown as a thin gray line). These three algorithms were sensitive to α and did not perform well. Emphatic TD(λ) reached an error of about 0.45, which was significantly higher than the error achieved by any other algorithm.

In this task, we divide the algorithms into three tiers. The top tier comprises Tree Backup(λ), Vtrace(λ), and ABTD(ζ) whose error was the lowest. The behavior of these was similar across Collision, Rooms, and High Variance Rooms tasks. In the middle tier are Off-policy TD(λ), GTD(λ), HTD(λ), and TDRC(λ). These algorithms achieved a slightly higher error than the top-tier algorithms but still reasonably solved the task. The bottom tier comprises Emphatic TD(λ), GTD2(λ), and Proximal GTD2(λ) whose best error level was even higher than second-tier algorithms.

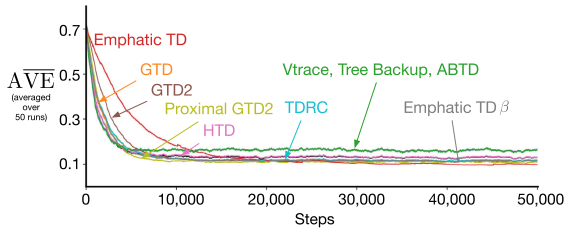


Fig. 8. Learning curves for the best algorithm instances of each learning algorithm in the Rooms task.

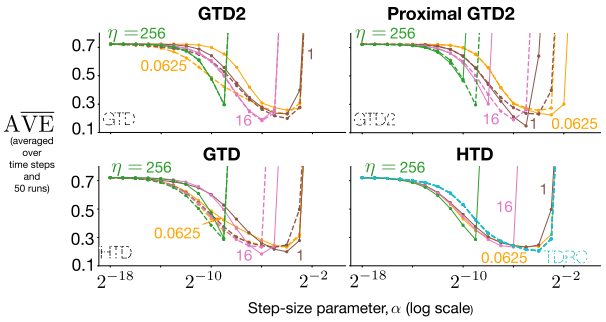


Fig. 9. Error as a function of α and η at $\lambda = 0$ on the Rooms task. Proximal GTD2 had the lowest error but was more sensitive to α than other algorithms. TDRC and HTD had the lowest sensitivity to α.

B. Emphatic-TD

We now set $\lambda = 0$ and study the behavior of Emphatic TD(λ, β) with varying β. Errors for various values of β are shown in the left figure of Fig. 11. The best performance was observed with small values of β. The bowl was nice and wide with $\beta = 0$ and $\beta = 0.2$. After that, with increasing β, the error consistently increased.

These results show that varying β significantly improves Emphatic TD(λ, β)'s performance. Without β, Emphatic TD(0) performed quite poorly on the High Variance Rooms task due to large variance. These results and the results from the Rooms task show that β's role becomes more salient as the problem variance increases. On the Collision task, no improvement was observed when varying β. On the Rooms task, intermediate values of β resulted in the best performance and, in the High Variance Rooms task, small values. The trend shows that as the problem variance increases, the magnitude of β that results in the best performance becomes smaller.

Learning curves for best algorithm instances of Emphatic TD(0) and Emphatic TD(0, β) are shown in the right figure of Fig. 11. These learning curves correspond to algorithm instances that resulted in minimum AUC. Emphatic TD(0, β) learned significantly faster than Emphatic TD. Emphatic TD(0) did not learn a reasonable approximation of the value function, probably due to being affected by the problem variance. The two dashed lines in the right figure of the figure show the approximate solutions for Emphatic TD(0) and Off-policy TD(0). Emphatic TD's solution had a significantly smaller $\overline{\text{AVE}}$ than Off-policy TD. This means that if the high variance was not present, Emphatic TD would find a solution with lower error than other algorithms such as Off-policy TD(λ).

In Rooms, High Variance Rooms, and Collision tasks, Emphatic TD had a lower asymptotic error than other algorithms. This has also been observed in some previous studies [42]. However, as the problem variance increases, Emphatic TD(λ) tends to learn slower. On the Collision task, it learned the fastest; on the Rooms task, it learned slower than other algorithms; it failed to learn a good approximation of the value function in the High Variance Rooms task. The

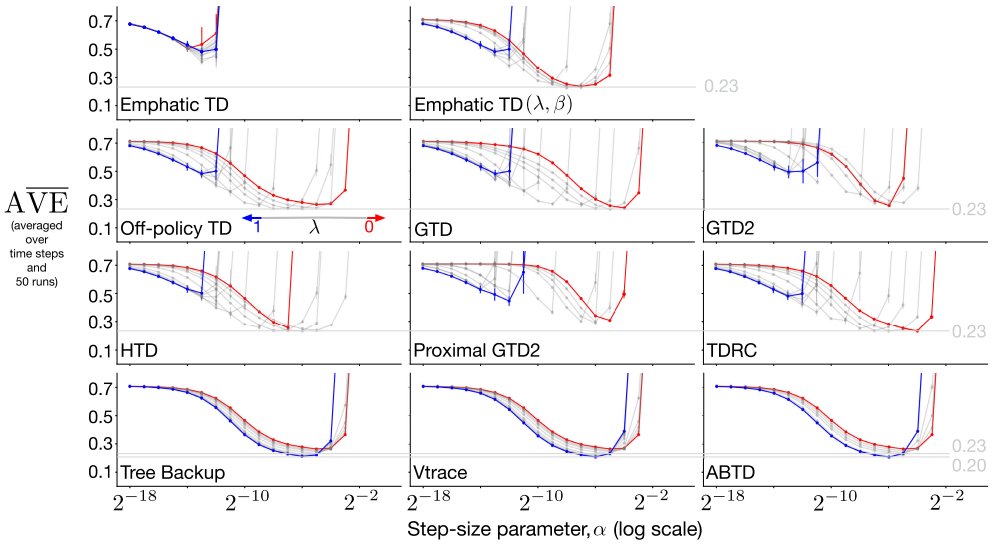


Fig. 10. Error as a function of α and λ for all algorithms on the High Variance Rooms task. Tree Backup(λ), Vtrace(λ), and ABTD(ζ) reached the lowest error level (0.2) and were in the top tier. All other algorithms except for Emphatic TD(λ), Proximal GTD2(λ), and GTD2(λ) reached the 0.23 error level and were in the middle tier. Emphatic TD(λ), Proximal GTD2(λ), and GTD2(λ) had a higher error than the rest of the algorithms, were more sensitive to α, and were grouped into the bottom tier.

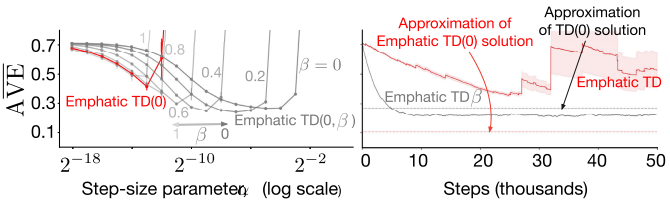


Fig. 11. Error as a function of α and β at $\lambda = 0$ on the High Variance Rooms task is shown on the left. The two best learning curves for Emphatic TD(λ, β) and Emphatic TD(λ) on the right show that Emphatic TD(λ, β) learned faster.

trend shows that Emphatic TD has a smaller asymptotic error across tasks but might overall be more prone to the variance issue than other algorithms. It also seems to be the case that the β parameter introduced by ETD(λ, β) helps mitigate the high-variance issue of ETD(λ). This is shown clearly as we move from the Collision task to the Rooms and High-Variance Rooms tasks.

C. Gradient-TD

We now turn to a more detailed analysis of Gradient-TD algorithms. Errors for $\lambda = 0$ for various η are plotted in Fig. 12.

According to the upper left figure, GTD2 generally performed worse than GTD. Based on the upper right figure, Proximal GTD2 had a significantly larger error than GTD2. According to the two lower figures, TDRC, HTD, and GTD all performed similarly and were relatively robust to the choice of α.

Let us now summarize the performance of Gradient-TD algorithms across tasks. On the Collision and Rooms tasks, Proximal GTD2 had the lowest error of all Gradient-TD algorithms. On the High Variance Rooms task, however, it had a higher error than all Gradient-TD algorithms. The trend across problems shows that Proximal GTD2 might be able to reach a lower error level than other Gradient-TD algorithms but is more prone to high variance than other Gradient-TD algorithms. In addition, the lower error level that it achieves does not seem to be of much practical utility because it is rare. GTD(λ), HTD(λ), and TDRC(λ) all seem to work well across tasks. HTD(λ) seems to be easier to tune across problems (see how its various bowls are smoother and wider than GTD in the lower left figure of Fig. 12). TDRC seems to be the easiest-to-use Gradient-TD algorithm because it has one tuned parameter and works, as well as HTD across tasks.

X. Large-Scale Experimental Results

The results on the Collision, Rooms, and High Variance Rooms environments suggest that: 1) emphatic TD(λ) tends to have a lower asymptotic error level, but it is more sensitive to the problem variance and 2) Tree Backup(λ), Vtrace(λ), and ABTD(ζ) are most robust to the problem variance. To examine whether this thesis holds in problems with larger state spaces as well, we designed an experiment where we gradually increased the size of the state space from 16 to 65 536.

We used empty gridworlds with 4×4, 8×8, 16×16, 32×32, and 256×256 states. Note that as the size of the state space increases, the product of the importance sampling ratios increases; as a result, the variance of the problem increases as well. The action space and the transition probabilities are the same as the Rooms task. The reward is 0 everywhere except upon transitioning to the bottom right cell where it is 1. The termination function returns 0 when transitioning to the bottom right cell. Otherwise, it returns $1 - (1/2x)$ where x is the square root of the size of the grid. This allows the expected time horizon of the prediction to be about $2x$, which is the number of steps that it takes to go from the top left state to the bottom right state. The target policy is to follow the shortest path to the bottom right state. The behavior policy is to go down or right with 0.27 probability and to go up and left with 0.23 probability. We used tile coding with one, two, four, eight, and 64 tilings each with 2×2 tiles for 4×4, 8×8, 16×16, 32×32, and 256×256 gridworlds, respectively.

The error for Emphatic TD(λ) and Vtrace(λ) relative to Off-policy TD(0) as a function of the size of the state space is shown in Fig. 13. A relative error of 1 suggests the same error level as Off-policy TD(0). The solid and dashed curves correspond to $\lambda = 0$ and $\lambda = 0.9$, respectively. Emphatic TD(λ) had a lower error than Off-policy TD(0) only in the 4×4 gridworld. As the size of the state space increased, the performance of Emphatic TD(λ) deteriorated drastically. Vtrace(λ) error level was similar to that of Off-policy TD(0) consistently as the size of the state space increased. These results confirm our conclusions from the previous sections

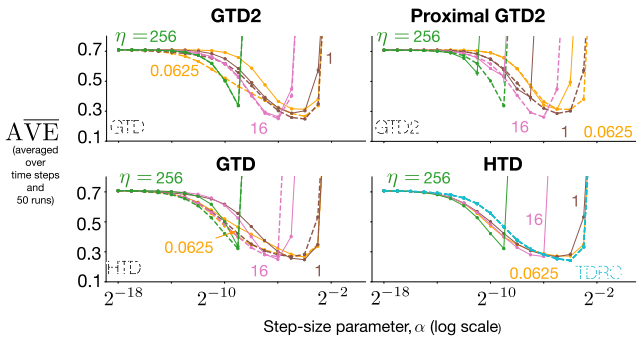


Fig. 12. Error as a function of α and η at $\lambda = 0$ on the High Variance Rooms task. The error of Proximal GTD2 (solid lines in the upper right figure) was higher than others.

suggesting that Emphatic TD(λ) tends to be more sensitive to the problem variance.

XI. Speeding Up Emphatic TD(λ)

This section introduces a step-size adaptation algorithm that increases Emphatic TD(λ)'s learning speed. Remember that Emphatic TD(λ) tends to have lower asymptotic error than other algorithms, but it can be slow on problems with high variance. The algorithms presented in this section adapt the step-size parameter of Emphatic TD(λ) and significantly increase its learning speed.

Two algorithms are introduced in this section: Step-size Ratchet and Soft Step-size Ratchet. The main idea behind both algorithms is to keep the step-size parameter as large as possible and ratchet it down when there is a possibility of overshoot. To show that the new algorithms are effective, we combine them with Emphatic TD(λ) and apply them to the Collision, Rooms, and High Variance Rooms tasks. Remember that Emphatic TD(λ) with constant step sizes could not learn a good approximation of the value function in the High Variance Rooms task and learned slowly on the Rooms task. We show that not only the combination of Emphatic TD(λ) and Ratchet algorithms learns a good approximation of the value function on all three problems but also they learn faster than the combination of Emphatic TD(λ) with other step-size adaptation algorithms such as Adam.

XII. Emphatic TD(λ) + Step-size Ratchet

In this section, we introduce the first step-size adaptation algorithm and combine it with Emphatic TD(λ). The main idea is to choose at each time step a step-size parameter that is as large as possible (or a fraction of it) without overshooting the target. Depending on how close the target of the update is to the current estimation of the value function, the step-size parameter is reduced only when necessary. The step-size parameter is never increased during learning.

At each time step, we compute the magnitude of the step-size parameter that, if used, will result in the estimate of the value function being equal to the target of the update. This means that we want to find the step size that, if used, the following equation would hold:

$$\mathbf{w}_{t+1}{}^\top \mathbf{x}_t = R_{t+1} + \gamma_{t+1}\mathbf{w}_t^\top \mathbf{x}_{t+1}. \tag{3}$$

In TD-style algorithms, the target of the update is the reward plus the discounted value of the next state. The update rules for Emphatic TD(λ) are

$$\begin{aligned} \delta_t &\overset{\text{def}}{=} \rho_t\Big(R_{t+1} + \gamma_{t+1}\mathbf{w}_t^\top \mathbf{x}_{t+1} - \mathbf{w}_t^\top \mathbf{x}_t\Big) \\ F_t &\leftarrow \rho_{t-1}\gamma_t F_{t-1} + I_t \quad \text{with } F_0 = I_0 \\ M_t &\overset{\text{def}}{=} \lambda_t I_t + \big(1-\lambda_t\big)F_t \end{aligned} \tag{4}$$

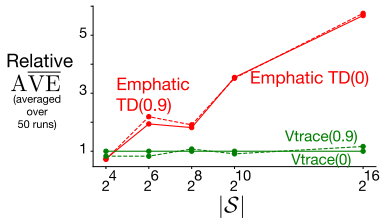


Fig. 13. Emphatic TD(λ) and Vtrace(λ) errors relative to Off-policy TD(0) as a function of the size of the state space.

$$\mathbf{z}_t \leftarrow \rho_{t-1}\gamma_t\lambda\mathbf{z}_{t-1} + M_t\mathbf{x}_t \quad \text{with } \mathbf{z}_{-1} = \mathbf{0} \tag{5}$$

$$\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t + \alpha_t\delta_t\mathbf{z}_t. \tag{6}$$

We now replace $\mathbf{w}_{t+1}$ on the left-hand side of (3) with the update rule provided for $\mathbf{w}_{t+1}$ in (6) and solve for α_t

$$\begin{aligned} (\mathbf{w}_t + \alpha_t\delta_t\mathbf{z}_t)^\top\mathbf{x}_t &= R_{t+1} + \gamma_{t+1}\mathbf{w}_t^\top\mathbf{x}_{t+1} \\ \alpha_t\delta_t\mathbf{z}_t^\top\mathbf{x}_t &= \underbrace{R_{t+1} + \gamma_{t+1}\mathbf{w}_t^\top\mathbf{x}_{t+1} - \mathbf{w}_t^\top\mathbf{x}_t}_{\frac{\delta_t}{\rho_t}} \\ \triangleright\alpha_t &= \frac{1}{\rho_t\mathbf{z}_t^\top\mathbf{x}_t}. \end{aligned} \tag{7}$$

We refer to the α_t that has the magnitude computed above in (7) as the *normalized step-size* parameter. We use a parameter κ to take a fraction of a normalized step toward the target at each time step: $\alpha_t = (\kappa/(\rho_t\mathbf{z}_t^\top\mathbf{x}_t))$.

It makes intuitive sense for the step-size parameter to be large at the beginning of learning and shrink down as learning goes on. The following minimization at each time step assures that the step-size parameter shrinks or remains the same

$$\alpha_t \leftarrow \min\left(\alpha_{t-1}, \frac{\kappa}{\rho_t\mathbf{z}_t^\top\mathbf{x}_t}\right). \tag{8}$$

At the first time step, α_{t-1} is set to a large number, maybe ∞. This completes the specification of the Step-size Ratchet algorithm. (See Appendix A (Supplementary Material) for the complete update rules of Emphatic TD(λ) augmented with Step-size Ratchet.)

XIII. Emphatic TD(λ) + Soft Step-size Ratchet

In this section, we introduce the second step-size adaptation algorithm Soft Step-size Ratchet and combine it with Emphatic TD(λ). The goal of Soft Step-size Ratchet is to relax the strict requirement of Step-size Ratchet that the step-size parameter can only become smaller or remain the same size at each time step. We will show that we are able to increase the magnitude of the step-size parameter at times while maintaining the main idea of the Step-size Ratchet algorithm. We show that this change will result in increasing the learning speed.

The Soft Step-size Ratchet algorithm works as follows. At each time step, first, the step-size parameter is computed using (8); the same update rule is used for Step-size Ratchet. Second, the step-size parameter, α_t, is used to update the weight vector $\mathbf{w}_t$, again the same as what the Step-size Ratchet algorithm does. The Step-size Ratchet algorithm, at this point, will go back to the first step and continue from there. The Soft Step-size Ratchet algorithm, instead, uses α_t and α_{t-1} to compute a new α_{t-1} that will be used in the next round of the updates once the execution of the algorithm goes back to the first step. This means that the only difference between the Step-size Ratchet and Soft Step-size Ratchet algorithms is that the α_{t-1} used in (8) will be updated before moving on to the next time step, pretending that the previous step-size

parameter was, in fact, larger than what was used to update $\mathbf{w}$. The update rule that we use for computing the new α_{t-1} is

$$\alpha_{t-1} \leftarrow \alpha_t + (\alpha_{t-1} - \alpha_t) \times \tau \tag{9}$$

where τ is a tunable parameter in (0, 1), which we set to 0.5 in all our experiments. After executing (9), the agent moves on to the next time step, goes back to the first step, and continues from there.

Let us now examine (9) more closely. Remember that before executing (9), the step-size parameter, α_t, is calculated using (8) at each time step, from which it immediately follows that $\alpha_{t-1} \geq \alpha_t$ for $\forall t$. It then follows that the term $(\alpha_{t-1} - \alpha_t)$ in (9) is always greater than or equal to zero. Let us first consider the case when it is zero. The difference being zero means $\alpha_t = \alpha_{t-1}$, meaning that α_{t-1} will not change after executing (9) because the term $(\alpha_{t-1} - \alpha_t)$ is equal to zero. If the difference is not zero, it means that $\alpha_{t-1} > \alpha_t$, in which case the value of α_{t-1} after performing update (9) will become larger proportional to τ. Let us, for example, assume that $\alpha_{t-1} = 1$, and the new step-size parameter computed by (8) is $\alpha_t = 0.5$ and $\tau = 0.5$. In this case, α_{t-1} for the next round of updates will be $\alpha_{t-1} = 0.5 + (1 - 0.5) \times 0.5 = 0.75$.

The magnitude of increase in α_{t-1} is proportional to the difference between α_{t-1} and α_t and is also proportional to the magnitude of τ. At the first time step, we set $\alpha_{-1} = (\kappa/(\mathbf{z}_t^\top \mathbf{x}_t))$. We call this new algorithm *Soft Step-size Ratchet* because ratcheting down the step-size parameter is soft, in which the step-size parameter can sometimes become a little larger. (See Appendix A (Supplementary Material) for the complete update rules of Emphatic TD(λ) augmented with Soft Step-size Ratchet.)

A. Experimental Setup and the Results

We applied the combination of Emphatic TD(λ) with Step-size Ratchet and Soft Step-size Ratchet to the Collision, Rooms, and High Variance Rooms tasks. We will present the results on the Rooms and High Variance Rooms tasks in this section as the difference across the step-size adaptation algorithms is more pronounced in these tasks. (See Appendix K (Supplementary Material) for the results of the Collision task.) We included Emphatic TD(λ) with constant step size and Emphatic TD(λ) with multiple step-size adaptation methods for comparison: 1) AlphaBound; 2) Adam; 3) AdamW; 4) AMSGrad; and 5) AdaGrad. Often, we refer to these combinations by the step-size adaptation algorithm name, leaving out Emphatic TD(λ) from the name because it is common across all combinations.

The AlphaBound algorithm [43] is similar to the Step-size Ratchet algorithm in that it has a single step-size parameter (rather than a vector of step-size parameters one for each direction) that only decreases in value over the course of learning. The schedule by which the step-size parameter shrinks is different from the schedule used by Step-size Ratchet. The intuition behind the AlphaBound algorithm is to make sure that the TD error at time step t gets closer to 0 after each update. (The derivation of Emphatic TD(λ) with AlphaBound can be found in Appendix A (Supplementary Material).)

Adam, or adaptive moment estimation algorithm, is one of the most commonly used step-size adaptation algorithms used in deep reinforcement learning [44]. Adam is often used with NN function approximation to increase learning speed. It uses statistics from the gradient vector to compute the direction and size of the update, in each dimension of the space, at each time step. Adam computes a vector of step sizes at each time step, one step-size scalar for each element of the weight vector, whereas the Ratchet algorithms compute a universal step-size parameter at each time step, one scalar step size for all elements of $\mathbf{w}$. (See Appendix K (Supplementary Material) for the derivation of Emphatic TD(λ) with Adam.) There have been improvements to Adam resulting in the AMSGrad [45] and AdamW algorithms, which we include in our experiments. We also include AdaGrad [46], which, similar to Adam, uses statistics to compute the direction and size of the update.

All of the experimental setup remains the same as what was discussed before other than the parameters of the step-size adaptation algorithms, which we will discuss shortly.

Similar to Section V, we use the term algorithm instance to refer to an algorithm with a specific set of parameters. The parameters of algorithms that we tried included all combinations of two values of λ (0, 0.9), 19 values of α for constant step-size parameters ($\alpha = 2^{-x}$ where $x \in \{0, 1, 2, \ldots, 17, 18\}$), 19 values of α_0 for AlphaBound and Adam algorithms ($\alpha_0 = 2^{-x}$ where $x \in \{0, 1, 2, \ldots, 17, 18\}$), 19 values of κ for Step-size Ratchet and Soft Step-size Ratchet ($\kappa = 2^{-x}$ where $x \in \{0, 1, 2, \ldots, 17, 18\}$), one value of τ for Soft Step-size Ratchet ($\tau = 0.5$), and three values of β_1 and β_2 for Adam (0.9, 0.99, 0.999). Finally, for Adam, we set $\epsilon = 10^{-8}$.

The Rooms task results are shown in Fig. 14. The top row shows the learning curves, and the bottom row shows the parameter sensitivity curves.

Let us first focus on the learning curve for the full bootstrapping case shown on the upper left figure of Fig. 14. On the Rooms task, we see the positive effect of applying the step-size adaptation algorithms to Emphatic TD(λ) clearly. Emphatic TD(λ) with constant step sizes learned the slowest, followed by AlphaBound, Adam, and AdamW, and AMSGrad and AdaGrad. Emphatic TD(0) combined with the Step-size Ratchet algorithm learned faster than all the baselines, as shown in the upper left figure of Fig. 14. The Soft Step-size Ratchet algorithm learned the fastest. In the case of full bootstrapping, the Soft Step-size Ratchet algorithm converged to the lowest asymptotic error level, followed by the Step-size Ratchet algorithm. Interestingly, the worst asymptotic error level was observed when Adam was used.

Let us now move on to consider the learning curves for the minimal bootstrapping case shown in the upper right figure of Fig. 14. Similar to the full bootstrapping case, the Soft Step-size Ratchet algorithm was the fastest. These were followed by Step-size Ratchet, AdaGrad, and AMSGrad. Regarding the asymptotic error level, the results were similar to the full bootstrapping case. The Soft Step-size Ratchet converged to the lowest error level followed by Step-size Ratchet, AdaGrad, and AMSGrad. The asymptotic error level was similar when the step-size parameter was constant and when Adam was used. With the AlphaBound algorithm, the asymptotic error level was a little lower than constant but statistically significantly higher than Soft Step-size Ratchet and Step-size Ratchet algorithms.

Two parameter sensitivity plots are shown in the lower row of Fig. 14. The sweet spot for the ratchet algorithms was shifted to the right compared to other algorithms. These two algorithms had the largest sweet spot at linear scale.

Overall, Step-size Ratchet and Soft Step-size Ratchet seem to be effective in increasing the learning speed of Emphatic TD(λ) when applied to the Rooms task. Moreover, at the linear scale, they seem to have a larger sweet spot than other algorithms.

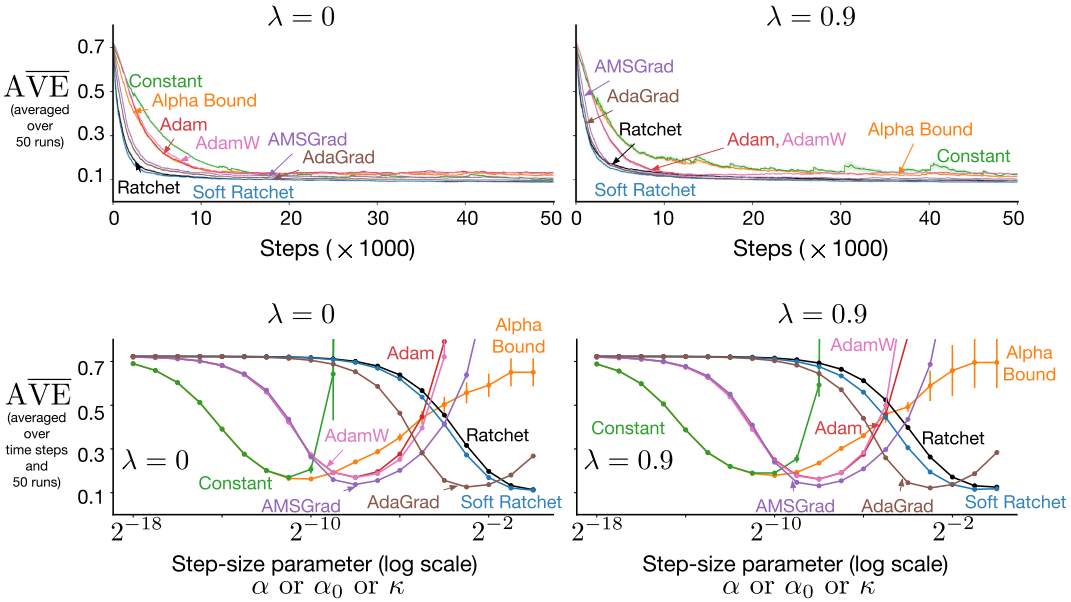


Fig. 14. Results of applying the combination of Emphatic TD(λ) and multiple step-size adaptation algorithms including Step-size Ratchet and Soft Step-size Ratchet to the Rooms task. The first row shows the learning curves, and the second row shows the parameter sensitivity curves. The Soft Step-size Ratchet algorithm learned the fastest and converged to the lowest error level followed by the Step-size Ratchet algorithm. Step-size Ratchet and Soft Step-size Ratchet algorithms had their U-shaped bowl shifted to the right compared to other algorithms. They had their minimum at around 2^{-2}.

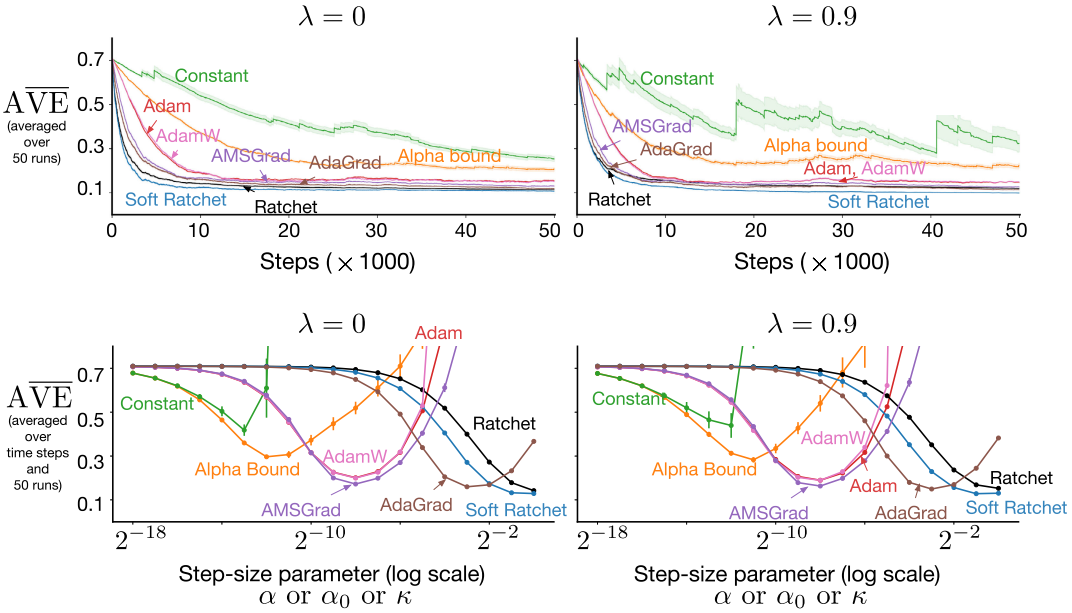


Fig. 15. Results of applying the combination of Emphatic TD(λ) and multiple step-size adaptation algorithms including Step-size Ratchet and Soft Step-size Ratchet to the High Variance Rooms task. The first row shows the learning curves, and the second row shows the parameter sensitivity curves. The Soft Step-size Ratchet algorithm learned the fastest followed by the Step-size Ratchet algorithm. The ratchet algorithms had their U-shaped bowl shifted to the right compared to other algorithms. They had their minimum at around 2^{-2}.

The High Variance Rooms task results are plotted in Fig. 15. The top row shows the best learning curves, and the bottom row shows the parameter sensitivity curves. Let us first focus on the two learning curves for full and minimal bootstrapping. The difference between the algorithms is more nuanced in the High Variance Rooms task than the Rooms task. Emphatic TD(λ) with a constant step-size parameter had difficulty learning the value function due to the high variance of the updates. Similar to the Rooms task, Soft Step-size Ratchet converged the fastest and to the lowest asymptotic error. Step-size Ratchet was the second fastest algorithm.

The parameter sensitivity curves show that all algorithms (except when constant step sizes were used) were equally sensitive to the value of their parameter at the logarithmic scale. Soft Step-size Ratchet had a lower error over a range of parameters compared to other algorithms.

Augmenting Emphatic TD(λ) with Step-size Ratchet and Soft Step-size Ratchet helped it reach Emphatic TD(λ)'s asymptotic solution. See Fig. 16; Emphatic TD(λ)'s asymptotic solution is shown with the horizontal dotted red line. As shown earlier, Emphatic TD(λ) with constant step size could not reach its asymptotic solution because the step size had to be set small due to the problem's high variance. The ratchet algorithms made learning substantially faster, making it possible for Emphatic TD(λ) to reach its asymptotic solution within 50 000 steps.

Overall, the Step-size Ratchet and Soft Step-size Ratchet algorithms were quite effective on the Rooms and High Variance Rooms tasks while maintaining the good performance of Emphatic TD(λ) on the Collision task. (For the results of the Collision task, see Appendix K (Supplementary Material).) The Soft Step-size Ratchet algorithm seems to be

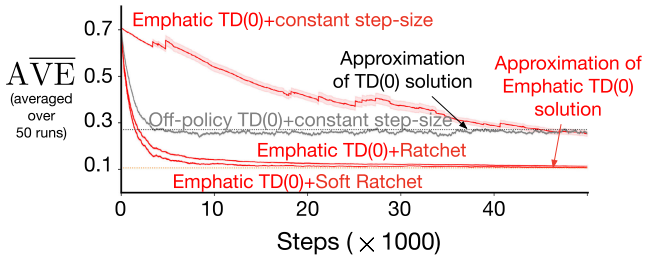


Fig. 16. Augmenting Emphatic TD(λ) with ratchet algorithms helped it reach its asymptotic solution. The dashed lines are approximate solutions of Emphatic TD(0) and TD(0), that is, the error level that these algorithms would converge to if they were applied to the task for long enough with a small step size.

the best algorithm across tasks consistently. On all tasks, the κ parameter that resulted in the lowest error for Step-size Ratchet and Soft Step-size Ratchet algorithms was around 2^{-2}. It is natural to ask whether it is easier to tune the parameters of the ratchet algorithms than the parameters of other algorithms across tasks. Given the data that we have from the three tasks, the answer seems to be yes. For a definite answer, more experimental results are, of course, necessary.

XIV. Conclusion and Limitations

This study paints a detailed picture of algorithms' performance as a function of the problem variance. Regarding the interplay of the algorithms' performance and variance of the problem, three main points were shown in this article for the first time.

1) Emphatic TD(λ) tends to have a lower asymptotic error level, but it is more prone to the high variance issue than other algorithms.
2) Proximal GTD2(λ) seems to be prone to the variance issue as well but less so than Emphatic TD(λ).
3) Tree Backup(λ), Vtrace(λ), and ABTD(ζ) are most robust to the problem variance but perform suboptimally on simple problems where high variance is not expected.

Our message for practitioners is to use Tree Backup, Vtrace, or ABTD wherever high variance is expected. For a task with moderate variance, an algorithm such as TD might be preferred if convergence is not of special importance. If convergence is of importance, an algorithm such as TDRC might be preferred. For a task with minimal variance, Emphatic TD might be preferred.

To achieve fast learning while benefiting from the lower asymptotic error of Emphatic TD(λ), we proposed two new step-size adaptation algorithms. We found that both algorithms are quite effective in speeding up Emphatic TD(λ)'s learning on the Rooms and High Variance Rooms tasks. We limited the application of Step-size Ratchet and Soft Step-size Ratchet algorithms to increase the learning speed of Emphatic TD(λ). However, not only do we expect the Ratchet algorithms to be applicable to other reinforcement learning algorithms but also we consider this a fruitful future research direction.

One limitation of the Soft Step-size Ratchet algorithm is that is has more than one tuned parameter. However, our experiments suggest that Soft Step-size Ratchet is robust to the choice of the τ parameter since with the one value of τ that we used in our experiments, Soft Step-size Ratchet performed well across tasks. It remains unclear how sensitive the Soft Step-size Ratchet algorithm is to the choice of the τ parameter and if tuning τ can provide a significant performance gain.

Another limitation of both the Step-size Ratchet and Soft Step-size Ratchet algorithms is that they are not readily applicable to nonstationary problems. These algorithms ratchet down the step-size parameter over time, which means that if the environment changes during learning, it might be hard for the agent to adapt to the new setting, depending on how small the step-size parameter is when the change happens.

Yet, another limitation of the proposed algorithms is that the step-size parameter that they compute at each time step is universal, meaning that a single step-size parameter is computed, which is used to update the parameter vector in all directions. Remember that, in principle, SGD is a component-wise process in which the step size for each component can be set separately. The Step-size Ratchet algorithm should ideally follow this principle, and as a result, it will be applicable to more flexible function approximators such as NNs.

Off-policy learning has come a long way but still has a long way to go. Two of the most central challenges of off-policy learning are stability and slow learning. The stability issue first became evident through Baird's counterexample [19]. Since then, Baird's counterexample has been used numerous times to exhibit various algorithms' stability in practice. In this article, we exhibited the variance challenge. The tasks introduced here can be used to assess algorithms' capability in learning in a high-variance setting. This article makes a step toward gaining a more granular understanding of the degree to which different off-policy prediction learning algorithms suffer from the problem of high variance and proposes two approaches for learning fast in the presence of high variance.

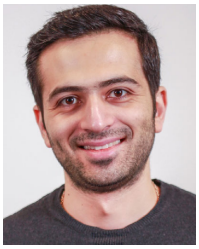
Acknowledgment

The authors would like to thank Ali Khalili for the help in preparing the source code. They also thank Martha White and Adam White for useful feedback throughout the course of this project. The computational resources of Compute Canada were essential to conducting this research.

References

[1] C. J. C. H. Watkins, "Learning from delayed rewards," Ph.D. dissertation, King's College, Univ. Cambridge, Cambridge, U.K., 1989.
[2] C. J. C. H. Watkins and P. Dayan, "Q-learning," *Mach. Learn.*, vol. 8, pp. 279–292, May 1992.
[3] V. Mnih et al., "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.
[4] H. Van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double Q-learning," in *Proc. AAAI Conf. Artif. Intell.*, 2016, vol. 30, no. 1, pp. 1–7.
[5] M. Hessel et al., "Rainbow: Combining improvements in deep reinforcement learning," in *Proc. AAAI Conf. Artif. Intell.*, 2018, vol. 32, no. 1, pp. 1–8.
[6] L. Espeholt et al., "IMPALA: Scalable distributed deeP-RL with importance weighted actor-learner architectures," in *Proc. Int. Conf. Mach. Learn.*, 2018, pp. 1407–1416.
[7] R. Jiang, S. Zhang, V. Chelu, A. White, and H. van Hasselt, "Learning expected emphatic traces for deep RL," in *Proc. AAAI Conf. Artif. Intell.*, 2022, vol. 36, no. 6, pp. 7015–7023.
[8] R. S. Sutton et al., "Horde: A scalable real-time architecture for learning knowledge from unsupervised sensorimotor interaction," in *Proc. 10th Int. Conf. Auto. Agents Multiagent Syst.*, vol. 2, 2011, pp. 761–768.
[9] A. White, "Developing a predictive approach to knowledge," Ph.D. dissertation, Dept. Comput. Sci., Univ. Alberta, Edmonton, AB, Canada, 2015.
[10] M. Ring, "Representing knowledge as predictions (and state as Knowledge)," 2021, *arXiv:2112.06336*.
[11] R. S. Sutton, D. Precup, and S. Singh, "Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning," *Artif. Intell.*, vol. 112, nos. 1–2, pp. 181–211, Aug. 1999.
[12] M. Littman and R. S. Sutton, "Predictive representations of state," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 14, 2001, pp. 1–14.
[13] B. Tanner and R. S. Sutton, "TD(λ) networks: Temporal-difference networks with eligibility traces," in *Proc. 22nd Int. Conf. Mach. Learn.*, 2005, pp. 888–895.
[14] M. Jaderberg et al., "Reinforcement learning with unsupervised auxiliary tasks," 2016, *arXiv:1611.05397*.
[15] P. S. Thomas, "Safe reinforcement learning," Ph.D. dissertation, Dept. Comput. Sci., Univ. Massachusetts Amherst, Amherst, MA, USA, 2015.

[16] Y. Xu and Z.-G. Wu, "Data-efficient off-policy learning for distributed optimal tracking control of HMAS with unidentified exosystem dynamics," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 35, no. 3, pp. 3181–3190, Mar. 2024.
[17] C. Sun, X. Li, and Y. Sun, "A parallel framework of adaptive dynamic programming algorithm with off-policy learning," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 32, no. 8, pp. 3578–3587, Aug. 2021.
[18] Y. Yang, Z. Guo, H. Xiong, D.-W. Ding, Y. Yin, and D. C. Wunsch, "Data-driven robust control of discrete-time uncertain linear systems via off-policy reinforcement learning," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 30, no. 12, pp. 3735–3747, Dec. 2019.
[19] L. Baird, "Residual algorithms: Reinforcement learning with function approximation," in *Machine Learning Proceedings*. Amsterdam, The Netherlands: Elsevier, 1995, pp. 30–37.
[20] D. Precup, R. S. Sutton, and S. Dasgupta, "Off-policy temporal-difference learning with function approximation," in *Proc. ICML*, 2001, pp. 417–424.
[21] H. R. Maei, "Gradient temporal-difference learning algorithms," Ph.D. dissertation, Univ. Alberta, Edmonton, AB, Canada, 2011.
[22] R. S. Sutton et al., "Fast gradient-descent methods for temporal-difference learning with linear function approximation," in *Proc. 26th Annu. Int. Conf. Mach. Learn.*, 2009, pp. 993–1000.
[23] L. C. Baird, "Reinforcement learning through gradient descent," Ph.D. dissertation, School Comput. Sci., Carnegie Mellon Univ. Pittsburgh, Pittsburgh, PA, USA, 1999.
[24] R. S. Sutton, A. R. Mahmood, and M. White, "An emphatic approach to the problem of off-policy temporal-difference learning," *J. Mach. Learn. Res.*, vol. 17, no. 73, pp. 1–29, 2016.
[25] S. Mahadevan et al., "Proximal reinforcement learning: A new theory of sequential decision making in primal-dual spaces," 2014, *arXiv:1405.6757*.
[26] S. Ghiassian, A. Patterson, S. Garg, D. Gupta, A. White, and M. White, "Gradient temporal-difference learning with regularized corrections," in *Proc. Int. Conf. Mach. Learn.*, 2020, pp. 3524–3534.
[27] A. Hallak, A. Tamar, R. Munos, and S. Mannor, "Generalized emphatic temporal difference learning: Bias-variance analysis," in *Proc. AAAI Conf. Artif. Intell.*, 2016, vol. 30, no. 1, pp. 1–7.
[28] R. Munos, T. Stepleton, A. Harutyunyan, and M. Bellemare, "Safe and efficient off-policy reinforcement learning," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 29, 2016, pp. 1–9.
[29] A. R. Mahmood, H. Yu, and R. S. Sutton, "Multi-step off-policy learning without importance sampling ratios," 2017, *arXiv:1702.03006*.
[30] K. Young and T. Tian, "MinAtar: An Atari-inspired testbed for thorough and reproducible reinforcement learning experiments," 2019, *arXiv:1903.03176*.
[31] M. Geist and B. Scherrer, "Off-policy learning with eligibility traces: A survey," *J. Mach. Learn. Res.*, vol. 15, pp. 289–333, Jan. 2014.
[32] C. Dann, G. Neumann, and J. Peters, "Policy evaluation with temporal differences: A survey and comparison," *J. Mach. Learn. Res.*, vol. 15, no. 1, pp. 809–883, 2014.
[33] A. White and M. White, "Investigating practical linear temporal difference learning," in *Proc. Int. Conf. Auto. Agents Multiagent Syst.*, 2016, pp. 494–502.
[34] W. Chung, S. Nath, A. Joseph, and M. White, "Two-timescale networks for nonlinear value function approximation," in *Proc. Int. Conf. Learn. Represent.*, 2018, pp. 1–32.
[35] D. Precup, R. Sutton, and S. Singh, "Eligibility traces for off-policy policy evaluation," in *Proc. 17th Int. Conf. Mach. Learn.*, 2000, pp. 759–766.
[36] R. S. Sutton, "Learning to predict by the methods of temporal differences," *Mach. Learn.*, vol. 3, no. 1, pp. 9–44, Aug. 1988.
[37] B. Liu, J. Liu, M. Ghavamzadeh, S. Mahadevan, and M. Petrik, "Proximal gradient temporal difference learning algorithms," in *Proc. IJCAI*, 2016, pp. 4195–4199.
[38] L. Hackman, "Faster gradient-TD algorithms," M.S. thesis, Dept. Comput. Sci., Univ. Alberta, Edmonton, AB, Canada, 2012.
[39] H. Yu, A. R. Mahmood, and R. S. Sutton, "On generalized Bellman equations and temporal-difference learning," *J. Mach. Learn. Res.*, vol. 19, no. 48, pp. 1–49, 2018.
[40] B. Rafiee, S. Ghiassian, A. White, and R. S. Sutton, "Prediction in intelligence: An empirical comparison of off-policy algorithms on robots," in *Proc. 18th Int. Conf. Auto. Agents MultiAgent Syst.*, 2019, pp. 332–340.
[41] J. Modayil and R. S. Sutton, "Prediction driven behavior: Learning predictions that drive fixed responses," in *Proc. Workshops 28th AAAI Conf. Artif. Intell.*, 2014, pp. 1–7.
[42] S. Ghiassian, B. Rafiee, and R. S. Sutton, "A first empirical study of emphatic temporal difference learning," 2017, *arXiv:1705.04185*.
[43] W. Dabney and A. Barto, "Adaptive step-size for online temporal difference learning," in *Proc. AAAI Conf. Artif. Intell.*, 2012, vol. 26, no. 1, pp. 872–878.
[44] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2014, *arXiv:1412.6980*.
[45] S. J. Reddi, S. Kale, and S. Kumar, "On the convergence of Adam and beyond," 2019, *arXiv:1904.09237*.
[46] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization," *J. Mach. Learn. Res.*, vol. 12, no. 7, pp. 2121–2159, 2011.
[47] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: MIT Press, 2018.
[48] R. S. Sutton, C. Szepesvári, and H. R. Maei, "A convergent O (n) algorithm for off-policy temporal-difference learning with linear function approximation," in *Proc. Adv. Neural Inf. Process. Syst.*, 2008, vol. 21, no. 21, pp. 1609–1616.
[49] A. Juditsky, A. Nemirovski, and C. Tauvel, "Solving variational inequalities with stochastic mirror-prox algorithm," *Stochastic Syst.*, vol. 1, no. 1, pp. 17–58, 2011.
[50] Y. Feng, L. Li, and Q. Liu, "A kernel loss for solving the Bellman equation," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 32, 2019, pp. 1–12.
[51] B. Dai et al., "SBEED: Convergent reinforcement learning with nonlinear function approximation," in *Proc. Int. Conf. Mach. Learn.*, 2018, pp. 1125–1134.
[52] X. Gu, S. Ghiassian, and R. S. Sutton, "Should all temporal difference learning use emphasis?" 2019, *arXiv:1903.00194*.
[53] M. White, "Unifying task specification in reinforcement learning," in *Proc. Int. Conf. Mach. Learn.*, 2017, pp. 3742–3750.
[54] A. Touati, P. L. Bacon, D. Precup, and P. Vincent, "Convergent tree backup and retrace with function approximation," in *Proc. Int. Conf. Mach. Learn.*, 2018, pp. 4955–4964.



Sina Ghiassian received the M.Sc. degree in supervised learning from the University of Alberta, Edmonton, AB, Canada, in 2014, and the Ph.D. degree from the University of Alberta in 2022, under the supervision of Richard S. Sutton.

He is currently a Research Scientist with Spotify Canada Inc., Toronto, ON, Canada. Before joining Spotify, he was a Post-Doctoral Fellow with the University of Alberta, where he was advised by Richard S. Sutton. During his Ph.D. degree, he worked on off-policy and emphatic temporal-difference learning algorithms.

Banafsheh Rafiee received the M.Sc. degree in computer science from the University of Alberta, Edmonton, AB, Canada, in 2018, where she is currently pursuing the Ph.D. degree.

Her research interest is in investigating the problem of representation learning in reinforcement learning.

Richard S. Sutton received the B.A. degree in psychology from Stanford University, Stanford, CA, USA, in 1978, and the Ph.D. degree in computer science from the University of Massachusetts, Boston, MA, USA, in 1984.

Prior to joining the University of Alberta, Edmonton, AB, Canada, in 2003, he worked in industry at AT&T Shannon Laboratory, Artificial Intelligence Department, Florham Park, NJ, USA and Computer and Intelligent Systems Laboratory, GTE Laboratories, Waltham, MA, USA, and in academia at the University of Massachusetts. He helped found DeepMind Alberta, Edmonton, AB, Canada, in 2017 and worked there until its dissolution in 2023. At the University of Alberta, he founded the Reinforcement Learning and Artificial Intelligence Laboratory, which now consists of ten principal investigators and about 100 people altogether. He is currently a Research Scientist with Keen Technologies, Westminster Ave, Dallas, USA, a Professor with the Department of Computing Science, University of Alberta, and a Chief Scientific Advisor of the Alberta Machine Intelligence Institute (Amii), Edmonton. He is a coauthor of the textbook *Reinforcement Learning: An Introduction*. His scientific publications have been cited more than 140 000 times. He is also a libertarian, a chess player, and a cancer survivor.

Dr. Sutton is a fellow of the Royal Society of London, the Royal Society of Canada, the Association for the Advancement of Artificial Intelligence, Amii, and Canadian Institute for Advanced Research (CIFAR).