SwiftTD: A Fast and Robust Algorithm for Temporal Difference Learning

Khurram Javed
kjaved@ualberta.ca

Arsalan Sharifnassab
sharifna@ualberta.ca

Richard S. Sutton
rsutton@ualberta.ca

Alberta Machine Intelligence Institute (Amii)
Department of Computing Science, University of Alberta
Edmonton, Canada

Abstract

Learning to make temporal predictions is a key component of reinforcement learning algorithms. The dominant paradigm for learning predictions from an online stream of data is Temporal Difference (TD) learning. In this work we introduce a new TD algorithm—SwiftTD—that learns more accurate predictions than existing algorithms. SwiftTD combines True Online TD(λ) with per-feature step-size parameters, step-size optimization, a bound on the update to the eligibility vector, and step-size decay. Per-feature step-size parameters and step-size optimization improve credit assignment by increasing the step-size parameters of important signals and reducing them for irrelevant signals. The bound on the update to the eligibility vector prevents overcorrections. Step-size decay reduces step-size parameters if they are too large. We benchmark SwiftTD on the Atari Prediction Benchmark and show that even with linear function approximation it can learn accurate predictions. We further show that SwiftTD performs well across a wide range of its hyperparameters. Finally, we show that SwiftTD can be used in the last layer of neural networks to improve their performance.

1 Temporal Difference Learning for Learning to Predict

Algorithms that can learn to predict the future are useful. Predicting the future is essential for sound decision-making, planning, and reasoning. An agent that wants to take the best action must be able to predict the values of different actions. Predicting the future is also a way to encode knowledge about the world. Unlike supervised learning which requires ground-truth labels, predictions can be learned solely from experience which makes predictive knowledge scalable.

A common issue when learning predictive knowledge is dealing with delayed feedback. Many predictions—such as *will it rain in two hours?*—require waiting for the predicted outcome to happen before the ground truth is available. The naive way to learn such predictions is to store the agent's experience and wait for the outcome. This scales poorly. An alternative is to use Temporal Difference (TD) learning and eligibility traces (Sutton, 1988).

TD learning is an online and scalable mechanism for learning predictive knowledge. It is also a crucial building block of many reinforcement learning algorithms, such as Sarsa(λ) (Rummery & Niranjan, 1994), Q-learning (Watkins & Dayan, 1992), PPO (Schulman et al., 2017), and Actor-Critic (Konda & Tsitsiklis, 1999). Improving TD learning has the potential to improve all these algorithms.

Existing algorithms for TD learning can be ineffective for learning incrementally and quickly. With existing algorithms, we are forced to make one of the following three unsatisfactory choices. First, we could learn with a small step-size parameter over a long period. This results in stable but slow learning. Second, we could attempt to learn with a large step-size parameter. Doing so could result

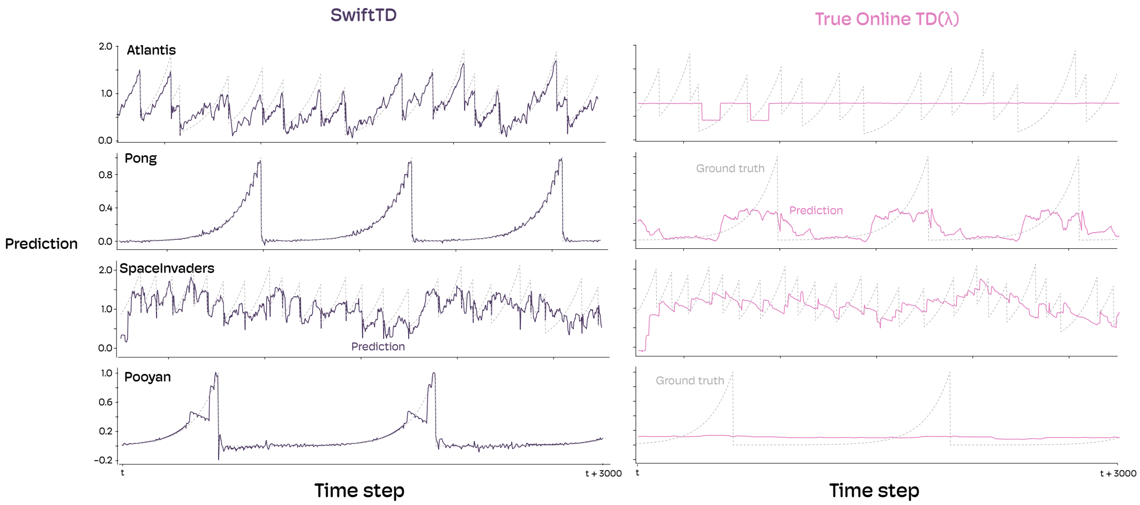

Figure 1: Predictions made by True Online TD(λ) and SwiftTD after learning for two hours of gameplay on Atari games. The gray dotted lines show the ground-truth returns. SwiftTD learned significantly more accurate predictions than True Online TD(λ). In some games—Pong, Pooyan—the predictions were near perfect. Even in more difficult games, like SpaceInvaders, the predictions anticipated the onset rewards.

in faster learning but risks divergence. Third, we could learn with a small step-size parameter but use every data point in multiple updates (*e.g.*, by using a replay buffer). The third choice allows sample efficient and robust learning and is used by popular Deep-RL algorithms (*e.g.*, see Mnih et al., 2015 and Schulman et al., 2017). However, doing multiple updates is computationally wasteful and leads to poor performance when learning in big worlds (Javed & Sutton, 2024). Moreover, requiring multiple updates for learning makes agents less reactive—feedback is not reflected in predictions and behaviors immediately.

Our goal with SwiftTD was to create a fourth option. A TD learner that could learn quickly, did not diverge, and did not require multiple updates to learn from feedback. Such an algorithm would allow a learner to learn as it is experiencing the data stream and remove the need for storing and replaying data.

Over two years we discovered three ideas that, when combined with True Online TD(λ), achieved our goal. Neither of the ideas was sufficient on its own to enable fast and robust learning. It is their unique combination that made SwiftTD work well.

The three ideas are: 1) step-size optimization for per-feature step-size parameters, 2) A bound on the increment to the eligibility vector to prevent overcorrections, and 3) a mechanism to selectively decay the step-size parameters if they are too large. Before we introduce the three ideas in detail, we provide a brief background of two algorithms that our ideas build upon.

2 Background

SwiftTD builds on two existing algorithms—True Online TD(λ) (Van Seijen et al., 2016) and Incremental Delta-bar Delta (IDBD) (Sutton, 1992). True Online TD(λ) uses λ-returns as targets for learning.

2.1 The λ-return

The λ-return is a form of multistep bootstrapped return that combines n-step returns for all n. The λ-return is defined as:

$$G_t^\lambda \overset{\text{def}}{=} (1-\lambda)\sum_{n=1}^{\infty}\lambda^{n-1}G_{t:t+n}, \tag{1}$$

where :

$$G_{t:t+n} \overset{\text{def}}{=} r_{t+1} + \gamma r_{t+2} + \cdots + \gamma^{n-1}r_{t+n} + \gamma^n v_{t+n}. \tag{2}$$

Here v_{t+n} is the agent's estimate of the sum of future discounted rewards at time $t+n$. λ-returns are unique because eligibility traces provide a computationally efficient mechanism to learn from them.

The simplest algorithm for learning with λ-returns is TD(λ) (Sutton, 1988). The weight updates done by TD(λ), however, are not exactly the same as an algorithm learning directly from λ-returns; they are only a good approximation when the step-size parameter is small.

2.2 True Online TD(λ)

True Online TD(λ) fixes the approximation error of TD(λ) and exactly matches the Online λ-return algorithm (Sutton & Barto, 2018) which does not use eligibility traces. Van Seijen et al. (2016) showed that True Online TD(λ) performs better than TD(λ) when learning with a large step-size parameter.

In later sections, we will show that using True Online TD(λ) instead of TD(λ) is crucial to achieving robust learning from large step-size parameters.

2.3 Incremental Delta-bar Delta (IDBD)

IDBD is an algorithm for meta-learning per-feature step-size parameters for linear regression. It uses gradient-based meta learning and incrementally approximates the gradients using forward-view differentiation (Williams & Zipser, 1989). Intuitively, IDBD increases the step-size parameters of features that generalize well to future examples and reduces them for features that generalize poorly.

IDBD is fundamentally different from popular adaptive step-size algorithms such as RMSProp (Tieleman & Hinton, 2012) and Adam (Kingma & Ba, 2015). Degris et al. (2024) argued that IDBD is doing *step-size optimization* when adapting the step-size parameters as opposed to RMSProp (Tieleman & Hinton, 2012), which is doing step-size normalization. They articulated the difference using a simple problem where IDBD adapted the step-size parameters to optimize a loss whereas RMSProp ignored the loss landscape when adapting the step-size parameters.

3 Formulating the Problem of Temporal Predictions

Our prediction problem consists of observations and predictions. The agent perceives an observation vector $\boldsymbol{\phi}_t \in \mathbb{R}^n$ at time step t and makes a scalar prediction v_t. The target for evaluating the prediction is computed by summing the future values of a scalar called the *cumulant*. The cumulant can be any scalar component of the observation vector. A common choice for the cumulant is the reward signal.

Performance on our prediction problem is measured by the *lifetime error*. Let r_t, a component of $\boldsymbol{\phi}_t$, be the cumulant at time t. The lifetime error is defined as:

$$\text{Lifetime error}(T) = \frac{1}{T}\sum_{t=1}^{T}\left(v_t - \sum_{j=t+1}^{T}\gamma^{j-t-1}r_j\right)^2, \tag{3}$$

where T is lifetime of the agent and γ is the discount factor. T and γ are part of the problem definition. The lifetime error captures not only the quality of the solution discovered by the agent at the end of learning, but also how quickly the agent finds the solution.

The lifetime error metric differs from the popular paradigm of splitting the data into disjoint train set and test set. Splitting the data is important in offline learning settings where the learner has access to the complete data set. It is unnecessary in online learning settings where the agent is evaluated on predictions made before getting the ground truth.

Before introducing the three ideas behind SwiftTD, we present a prediction benchmark in the next section that we use to explain the three ideas.

4 The Atari Prediction Benchmark

The Atari Prediction Benchmark (Javed et al., 2023) is a suite of prediction problems. It is built on the Arcade Learning Environment (ALE) (Bellemare et al., 2013), a collection of Atari 2600 games. In each game, a player can take up to 18 discrete actions with the goal to maximize the score. The Atari Prediction Benchmark constructs prediction problems from ALE by picking actions using pre-trained Rainbow-DQN (Hessel et al., 2018) policies taken from the model zoo of Chainer-RL (Fujita et al., 2021).

To convert the Atari Prediction Benchmark into a set of temporal prediction problems, as defined in Section 3, we have to specify the observation vector, the cumulant, the discount factor γ, and the lifetime of the agent (T) for each game.

We construct the observation vector of the agent by preprocessing the game frame and turning it into a binary valued vector as explained in the next subsection. We generate the cumulant by preprocessing the reward given by the Arcade Learning Environment. A positive reward from ALE sets the cumulant to +1, and a negative reward sets it to -1. The cumulant is zero if the reward is zero. We use $\gamma = 0.98$ for all games. For all experiment, we set the lifetime to be 210,000. This translates to roughly 2 hours of gameplay at 30 frames per second.

4.1 Constructing the feature vector from the game frame

The Atari game frame is a tensor of dimensions 210 × 160 × 3. Every component of this tensor is a scalar in the range [0, 255].

In the preprocessing steps, we first resize the frame to 105 × 80 × 3. We convert each of the three channels in the resized frame to a tensor of dimensions 105 × 80 × 8 by performing a lossy one-hot coding to the value of each pixel. Pixel values from 0 to 31 set the first channel to one and the remaining seven to zero, values from 32 to 63 set the second channel to one and the rest to zero, and so on. Figure 2 (a) illustrates the binning process with a simple example and Figure 2 (b) shows the binning process applied to a frame of the game Freeway.

The binning process gives us three tensors of dimensions 105 × 80 × 8. We stack them to get a tensor of dimensions 105 × 80 × 24 and flatten it to get a vector with 201,600 binary valued components. We then append the previous one-hot coded action (a vector with 18 components) and the cumulant to the 201,600 length vector to get the final feature vector with 201,619 components.

We use the Atari Prediction Benchmark with the above-mentioned preprocessing to compare different algorithms in all experiments except in Section 7, where we use a convolutional network to transform the 105 × 80 × 24 tensor into a vector.

5 Three Ideas for Fast and Robust TD Learning

SwiftTD is a combination of three ideas—step-size optimization for per-feature step-size parameters, a bound on the update to the eligibility vector, and step-size decay for per-feature step-size

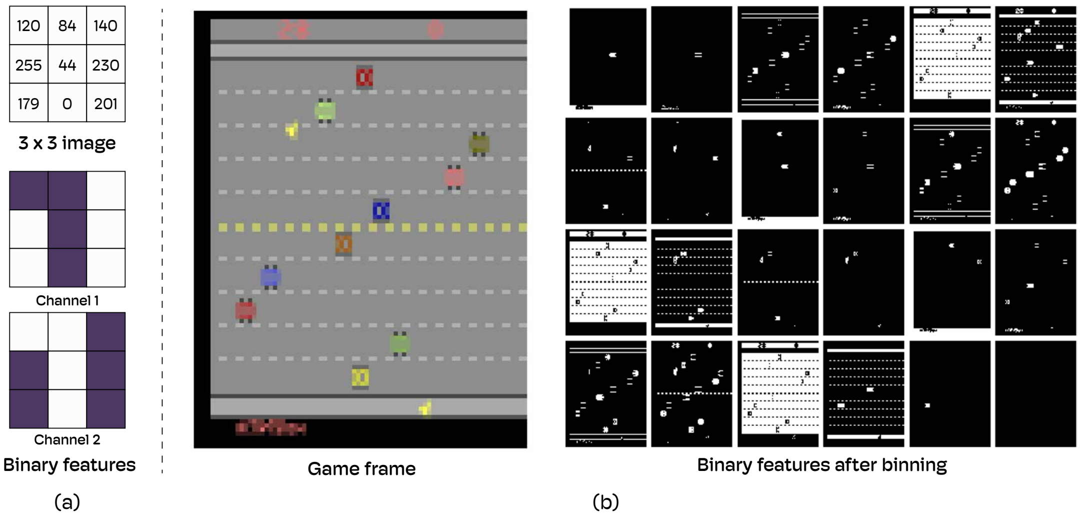

Figure 2: **(a)** A simplified example of the binning step with a 3 × 3 image. We transform the image into a binary valued tensor by binning the value of the pixel into two channels. Pixel values from 0 to 127 are binned into the first channel, and 128 to 255 into the second channel. **(b)** The binning process applied to a real frame on the game Freeway. In our experiments, the agent learns from the binary features generated by the binning process.

parameters. In the following subsections, we introduce each idea separately. We then combine them to develop SwiftTD in the next section.

Each agent in our algorithms is parameterized by a weight vector $\boldsymbol{w} \in \mathbb{R}^n$ and makes predictions by linearly combining the features with the weights as:

$$v_t = \sum_{i=1}^{n} \boldsymbol{w}_{t-1}[i]\boldsymbol{\phi}_t[i], \tag{4}$$

where $\boldsymbol{\phi}_t[i]$ is the ith component of $\boldsymbol{\phi}_t$. The time index for the weight vector is $t-1$ because the prediction is made before the weight vector is updated using $\boldsymbol{\phi}_t$.

5.1 Idea 1: TD Learning with Step-size Optimization

An immediate implication of our goal to learn quickly in a single update is that we must do large updates to the parameters. At the same time, an implication of being robust is that we must not do large updates to weights associated with noisy or irrelevant features. The two requirements are at odds if we limit ourselves to using a scalar step-size parameter. However, if we allow the agent to have different step-size parameters for different weights, both requirements could be simultaneously satisfied.

If we commit to using different step-size parameters for different weights, the next important question is how to set the parameters. We can no longer set them manually as the agent can have millions of weights. A viable option is to learn the step-size parameters from experience.

Using learnable per-feature step-size parameters is precisely the first key idea behind SwiftTD. To learn them, we parameterize the step-size parameters with a vector $\boldsymbol{\beta} \in \mathbb{R}^n$ and use $\boldsymbol{\alpha}[i] = e^{\boldsymbol{\beta}[i]}$ in weight updates.

We can learn the step-size parameter $\boldsymbol{\beta}[i]$ by updating it as:

$$\boldsymbol{\beta}_{\boldsymbol{t}}[i] = \boldsymbol{\beta}_{\boldsymbol{t-1}}[i] - \frac{\theta}{e^{\boldsymbol{\beta}[i]}} \frac{\partial \left(v_t - G_t^{\lambda}\right)^2}{\partial \boldsymbol{\beta}[i]}, \tag{5}$$

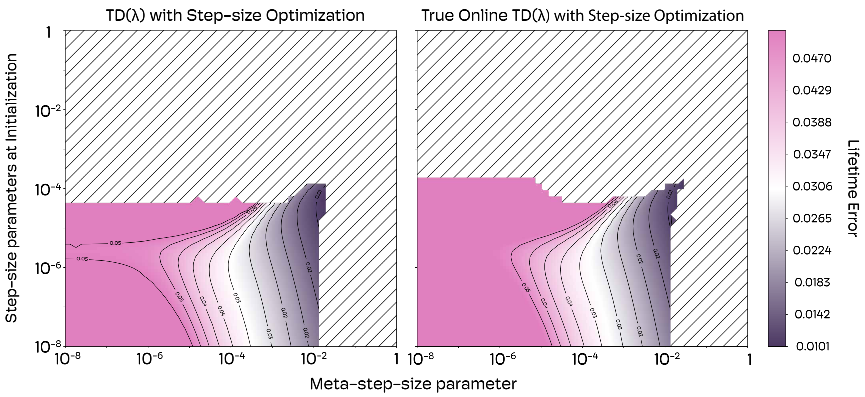

Figure 3: Impact of step-size optimization on the lifetime error of TD(λ) and True Online TD(λ) for different values of step-size parameters at initialization (α^{init}) and meta-step-size parameter (θ). All finite errors above 0.5 are clipped to 0.5. Points where the lifetime error diverged are shown with diagonal lines. Increasing the meta-step-size improved the performance of both algorithms up to a point. This suggests that step-size optimization for per-feature step-size parameters can help improve the performance of TD learning.

where $\frac{\theta}{e^{\boldsymbol{\beta}[i]}}$ is the meta-step-size parameter [1].

Computing the exact meta-gradient is computationally expensive. We instead use an approximation of the meta-gradient similar to IDBD. We provide the derivations and pseudocode of TD(λ) with step-size optimization and True Online TD(λ) with step-size optimization in Appendix A.

5.1.1 Related work

We are not the first to suggest step-size optimization for TD learning. Three prior works have attempted to extend IDBD to TD learning.

Two of them—by Thill (2015) and Kearney et al. (2018)—incorrectly estimated the meta-gradient. Thill (2015) made a mistake when deriving the update rule for the meta-gradient. Kearney et al. (2018) derived the meta-gradient correctly, but used the TD(0) objective for the meta-gradient even when learning with TD(λ). The discrepancy is problematic and can fail to increase step-size parameters of features correctly (Javed, 2024).

Young et al. (2019) correctly extended IDBD to TD(λ). After we developed SwiftTD, we found that our extension of IDBD to TD(λ) was identical to that by Young et al. (2019). Our extension of IDBD to True Online TD(λ) is novel to this work.

5.1.2 Experiments and results: The impact of step-size optimization on the lifetime error

TD learning with step-size optimization has two key hyperparameters—the meta-step-size parameter (θ) and the value of the step-size parameters at initialization (α^{init}). To understand the usefulness of step-size optimization, we measure the lifetime error of these algorithms as a function of θ and α^{init}. We use the Pong environment from the Atari Prediction Benchmark.

We ran both algorithms for fifty-five values of α^{init} and θ on the game Pong. For both parameters, we used values from the set $\{0.7^x | x \in \{0, 1, \cdots, 54\}\}$. This translates to a total of 3025 experiments.

We plot all results in a single 2D plot in Figure 3 where the x-axis is the meta-step-size parameter, the y-axis is the initial value of the step-size parameters, and the color represents the lifetime error.

[1] We normalize θ by $e^{\boldsymbol{\beta}[i]}$ because the scale of the meta-gradient of the error with respect to $\boldsymbol{\beta}[i]$ is proportional to $e^{\boldsymbol{\beta}[i]}$

The performance of both algorithms improved on Pong as the meta-step-size parameter increased from 10^{-8} up to 10^{-2}. It diverged when the meta-step-size went over 10^{-2}. This trend held for α^{init} in the range 10^{-8} to 10^{-5}. The algorithm also diverged for α^{init} larger than 10^{-4}

The three main conclusions from the sensitivity analysis are 1) step-size optimization can help achieve lower error, 2) TD learning with step-size optimization can diverge when its hyperparameters are chosen poorly, and 3) TD(λ) and True Online TD(λ), when combined with step-sized optimization, perform similarly.

The divergence when learning from certain values of θ and α^{init} is a significant problem. It is unlikely that the hyperparameters that worked well for Pong would also work well for other problems. There is a need for a mechanism that can prevent divergence.

5.2 Idea 2: TD Learning with the Overshoot Bound

The primary issue with TD learning with step-size optimization is that if the step-size parameters are initialized to be too large, the learner diverges. Moreover, even for a small initial value of the step-size parameters, if the meta-step-size parameter is too large, the step-size parameters can eventually get too large due to the meta-learning updates. To fix the problem of divergence, we introduce a bound on the update to the eligibility vector called the *overshoot bound*.

We define *correction ratio* of an update as a measure of how close a prediction is to the target after the update. Let y_t^* be the target for the prediction at time t. The correction ratio of a weight update for a linear learner is the fraction of the error reduced after the update. More precisely, we define it as:

$$\tau_t = \frac{(y_t^* - \sum_{i=1}^{n} \boldsymbol{w}_{t-1}[i]\boldsymbol{\phi}_t[i]) - (y_t^* - \sum_{i=1}^{n} \boldsymbol{w}_{t'}[i]\boldsymbol{\phi}_t[i])}{(y_t^* - \sum_{i=1}^{n} \boldsymbol{w}_{t-1}[i]\boldsymbol{\phi}_t[i])}, \tag{6}$$

where $\boldsymbol{w}_{t'}$ is the weight vector after the updates. For linear regression with per-component step-size parameters $\boldsymbol{\alpha}_t$, the correction ratio can be simplified to:

$$\tau_t = \sum_i \boldsymbol{\alpha}_t[i]\boldsymbol{\phi}_t[i]^2. \tag{7}$$

A correction ratio of 1.0 means that the prediction has changed to the target. A correction ratio of 0.5 means that the prediction has changed to halfway between the old prediction and the target.

If we can bound the correction ratio of every update to be less than or equal to one, we can guarantee that the prediction with the updated weights does not overshoot the target. This is precisely the idea behind the overshoot bound. This idea is not new. Mahmood et al. (2012) proposed a similar bound for linear regression. Our contribution is a mechanism to implement this bound for TD learning.

The two challenges in applying this bound to TD learning are: 1) dealing with delayed targets, and 2) dealing with fact that our targets also depend on the weights of the agent.

We address the first challenge by realizing that True Online TD(λ) has perfect equivalence with a learner that does not use delayed targets (The Online λ-return algorithm). This makes it possible to apply the bound to True Online TD(λ). Care must be taken when applying the bound. It is not possible to apply the bound at the time of the parameter update. Rather, it must be applied when adding to the eligibility vector as shown in Algorithm 6.

We address the second challenge by ignoring the influence of the weights on the targets (the semi-gradient assumption). This makes sense as the goal in TD learning is not to minimize the error but to propagate credit to the correct features by matching predictions to targets. The pseudocode and derivations of the overshoot bound for True Online TD(λ) and TD(λ) are in Appendix B.

5.2.1 Related work

Dabney and Barto (2012) made an attempt to bound the correction ratio of TD learning. Their bound did not make the semi-gradient assumption and used the one-step bootstrapped target as

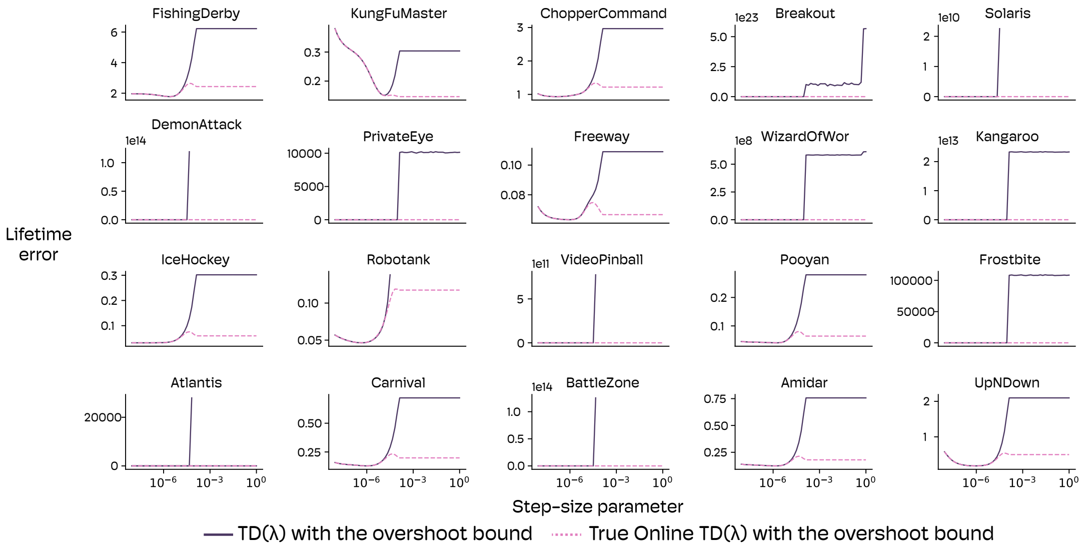

Figure 4: Impact of α on the lifetime error for TD(λ) with the overshoot bound and True Online TD(λ) with the overshoot bound. The bound was not successful for TD(λ) and the lifetime error diverged for large step-size parameters on multiple games. True Online TD(λ) with the bound, on the other hand, did not diverge on any of the games and performed reasonably well even for very large step-size parameters.

opposed to the λ-return target. We tried their bound and found that it did not fix the divergence issue. Their bound can allow arbitrarily large changes to the weights of an agent if the features for consecutive time steps are highly correlated.

5.2.2 Experiments and results: The impact of the overshoot bound on the lifetime error

We implemented the bound as shown in Algorithm 4 and Algorithm 6 and plot the lifetime error of both algorithms on a randomly selected subset of games in the Atari Prediction Benchmark as a function of α in Figure 4.

True Online TD(λ) with the overshoot bound did not diverge for any values of α. Moreover, the bound was not conservative and did not change the performance for the best value of α. It only kicked in once α was larger than the best α. TD(λ) with the overshoot bound, on the other hand, performed poorly in some games (*e.g.*, Frostbite, Kangaroo, and Breakout) and diverged in some games (*e.g.*, VideoPinball, DemonAttack, and Battlezone).

While the overshoot bound prevents divergence, it being triggered is a sign that the step-size parameters are too large. We can combine the bound with step-size optimization and hope that step-size optimization will reduce the step-size parameters when they are too large. However, the way the bound is applied is not a differentiable operation and gradient-based step-size optimization cannot be applied. Our third key idea is to force the step-size parameters to get smaller whenever the bound is triggered. We call this idea *step-size decay*.

5.3 Idea 3: TD Learning with the Overshoot Bound and Step-size Decay

If we know that the step-size parameters are too large, we don't have to rely on the meta-gradient to adapt them. We can simply reduce them.

Mechanistically, step-size decay is simple to implement. Let $\boldsymbol{\alpha}_t$ be the step-size parameters at time t. At every step for which the bound is active because the overcorrection ratio is greater than 1, we

```
Algorithm 1: SwiftTD
Parameters with default values: $\epsilon = 0.99, \eta = 0.1, \eta^{min} = e^{-15}, \alpha^{init} = 10^{-7}, \lambda, \gamma, \theta$
Initializations: $\boldsymbol{w}, \boldsymbol{h^{old}}, \boldsymbol{h^{temp}}, \boldsymbol{z^{\delta}}, \boldsymbol{p}, \boldsymbol{h}, \boldsymbol{z}, \bar{\boldsymbol{z}} \leftarrow \boldsymbol{0} \in \mathbb{R}^n; (v^{\delta}, v^{old}) = (0, 0); \boldsymbol{\beta} \leftarrow \ln(\boldsymbol{\alpha}^{init}) \in \mathbb{R}^n$
while alive do
    Perceive $\boldsymbol{\phi}$ and $r$
    $v \leftarrow \sum_{\boldsymbol{\phi}[i] \neq 0} \boldsymbol{w}[i]\boldsymbol{\phi}[i]$
    $\delta' \leftarrow r + \gamma v - v^{old}$
    for $\boldsymbol{z}[i] \neq 0$ do
        $\boldsymbol{\delta^w}[i] \leftarrow \delta' \boldsymbol{z}[i] - \boldsymbol{z^{\delta}}[i] v^{\delta}$
        $\boldsymbol{w}[i] \leftarrow \boldsymbol{w}[i] + \boldsymbol{\delta^w}[i]$
        $\boldsymbol{\beta}[i] \leftarrow \boldsymbol{\beta}[i] + \frac{\theta}{e^{\boldsymbol{\beta}[i]}}(\delta' - v^{\delta})\boldsymbol{p}[i]$
        $\boldsymbol{\beta}[i] \leftarrow \text{clip}\left(\boldsymbol{\beta}[i], \ln(\eta^{min}), \ln(\eta)\right)$        // Clip $\beta$ to be in range [ln($\eta^{min}$), ln($\eta$)]
        $\boldsymbol{h^{old}}[i] \leftarrow \boldsymbol{h}[i]$
        $\boldsymbol{h}[i] \leftarrow \boldsymbol{h^{temp}}[i]$
        $\boldsymbol{h^{temp}}[i] \leftarrow \boldsymbol{h}[i] + \delta' \bar{\boldsymbol{z}}[i] - \boldsymbol{z^{\delta}}[i] v^{\delta}$
        $\boldsymbol{z^{\delta}}[i] = 0$
        $(\boldsymbol{z}[i], \boldsymbol{p}[i], \bar{\boldsymbol{z}}[i]) \leftarrow (\gamma\lambda\boldsymbol{z}[i], \gamma\lambda\boldsymbol{p}[i], \gamma\lambda\bar{\boldsymbol{z}}[i])$
    $v^{\delta} \leftarrow 0$
    $\tau \leftarrow \sum_{\boldsymbol{\phi}[i] \neq 0} e^{\boldsymbol{\beta}[i]} \boldsymbol{\phi}[i]^2$
    $T \leftarrow \sum_{\boldsymbol{\phi}[i] \neq 0} \boldsymbol{z}[i]\boldsymbol{\phi}[i]$
    for $\boldsymbol{\phi}[i] \neq 0$ do
        $v^{\delta} \leftarrow v^{\delta} + \boldsymbol{\delta^w}[i]\boldsymbol{\phi}[i]$
        $\boldsymbol{z^{\delta}}[i] \leftarrow \min\left(1, \frac{\eta}{\tau}\right) e^{\boldsymbol{\beta}[i]} \boldsymbol{\phi}[i]$        // Overshoot bound
        $\boldsymbol{z}[i] \leftarrow \boldsymbol{z}[i] + \boldsymbol{z^{\delta}}[i](1 - T)$
        $\boldsymbol{p}[i] \leftarrow \boldsymbol{p}[i] + \boldsymbol{\phi}[i]\boldsymbol{h}[i]$
        $\bar{\boldsymbol{z}}[i] \leftarrow \bar{\boldsymbol{z}}[i] + \boldsymbol{z^{\delta}}[i]\left(1 - T - \boldsymbol{\phi}[i]\bar{\boldsymbol{z}}[i]\right)$
        $\boldsymbol{h^{temp}}[i] \leftarrow \boldsymbol{h^{temp}}[i] - \boldsymbol{h^{old}}[i]\boldsymbol{\phi}[i]\left(\boldsymbol{z}[i] - \boldsymbol{z^{\delta}}[i]\right) - \boldsymbol{h}[i]\boldsymbol{z^{\delta}}[i]\boldsymbol{\phi}[i]$
        if $\tau > \eta$ then
            $\boldsymbol{\beta}[i] = \boldsymbol{\beta}[i] + \boldsymbol{\phi}[i]^2 \ln(\epsilon)$        // Step-size decay
            $(\boldsymbol{h^{temp}}[i], \boldsymbol{h}[i], \bar{\boldsymbol{z}}[i]) = (0, 0, 0)$
    $v^{old} \leftarrow v$
```

update the step-size parameters as:

$$\boldsymbol{\alpha}_{t+1}[i] = \boldsymbol{\alpha}_t[i]\epsilon^{\boldsymbol{\phi}_t[i]^2}, \tag{8}$$

where ϵ is a hyperparameter called the decay rate. A reasonable choice for ϵ is 0.99. Note that we are not decaying the step-size parameter for all features. We are only decaying the step-size parameter proportional to the squared value of the features. Pseudocode for True Online TD(λ) with step-size decay is in Algorithm 7.

5.3.1 Related work

The idea of step-size decay is similar in spirit to the Step-size Ratchet algorithm proposed by Ghiassian (2022). It differs from Step-size Ratchet in three important ways. First, Step-size Ratchet decays the step-size parameters abruptly to satisfy its bound as opposed to decaying them slowly. Second, it uses the one-step bootstrapped target as opposed to the λ-return for computing the overcorrection ratio. Finally, it uses a scalar step-size parameter and does not decay step-size parameters proportional to their contribution to the overcorrection ratio.

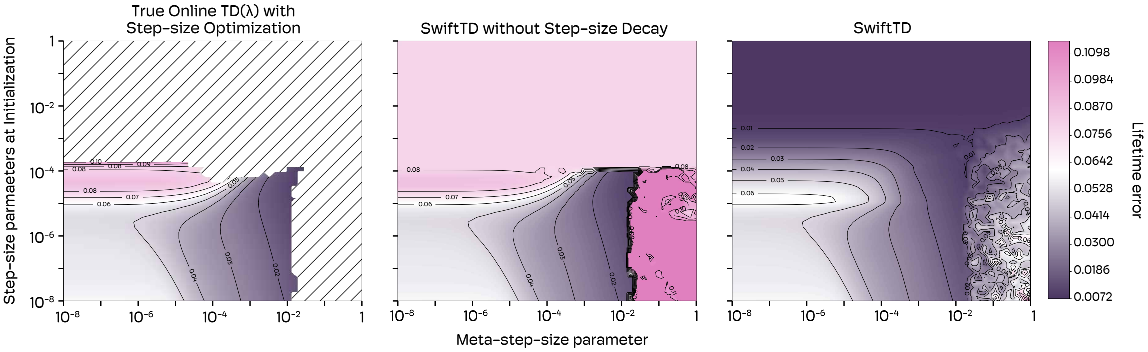

Figure 5: Performance of SwiftTD on Pong for different α^{init} and θ. Here SwiftTD used $\eta = 0.1$ and $\epsilon = 0.999$. It did not diverge for any combination of α^{init} and θ and performed reasonably well for almost all combinations. SwiftTD without step-size decay also did not diverge but performed poorly for large meta-step-size parameters and initial step-size parameters.

6 SwiftTD: Fast and Robust Learning by Combining the Three Ideas

Now that we have explained the three ideas separately, we combine them in a single algorithm called SwiftTD. We make two additional changes. First, we generalize the idea of overshoot bound by introducing a new hyperparameter η. In SwiftTD, the overshoot bound is triggered whenever the correction ratio is greater than η instead of one. A reasonable default value for η is 0.1. Second, at every step, we clip every step-size parameter to be in range $[\ln(\eta^{min}), \ln(\eta)]$.

Algorithm 1 is the pseudocode for SwiftTD. The pink lines implement step-size optimization, the blue lines implement the overshoot bound, the purple lines implement step-size decay, and the orange line implements the clipping of the step-size parameters. The remaining black lines are the same as True Online TD(λ).

Intuitively, SwiftTD increases the step-size parameters of relevant features and reduces them for irrelevant features. If the step-size parameters become too large, it uses the overshoot bound to prevent bad updates while simultaneously decaying the step-size parameters proportional to their contribution to the correction ratio.

To demonstrate the effectiveness of SwiftTD we did the hyperparameter sensitivity analysis of SwiftTD for α^{init} and θ on Pong and report the results in Figure 5. We used $\eta = 0.1$ and $\epsilon = 0.999$ for the right most plot. SwiftTD was not only stable for all combinations of α^{init} and θ but also performed reasonably well for almost all combinations. SwiftTD without step-size decay, on the other hand, performed poorly when either the initial value of the step-size parameters or the meta-step-size parameter was too large. Figure 5 also demonstrates that all three ideas are needed for the strong and robust performance of SwiftTD.

6.1 Hyperparameter Tuning and Results on All Games

We compared SwiftTD and True Online TD(λ) on all Atari games. For both SwiftTD and True Online TD(λ), we swept over all their hyperparameters. Because SwiftTD has more hyperparameters than True Online TD(λ), we did a coarser search over its hyperparameters for a fair comparison. The details of the hyperparameter sweeps are in Appendix D.

We tuned all hyperparameters for each Atari game individually and report the results with the best hyperparameter setting for each game. An alternative choice would have been to tune the hyperparameters on a subset of the games and use the same hyperparameters for all games. Both choices have their advantages and disadvantages. We verified that the results did not change qualitatively with either choice.

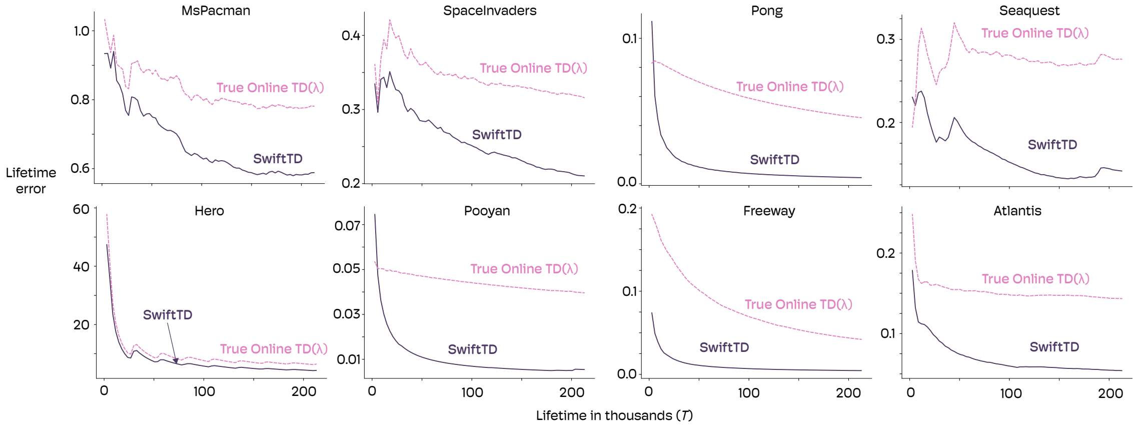

Figure 6: Learning curves for eight games. The y-axis is the lifetime error and the x-axis is the lifetime. In all games, SwiftTD had lower lifetime error than True Online TD(λ) for almost all values of the lifetime parameter.

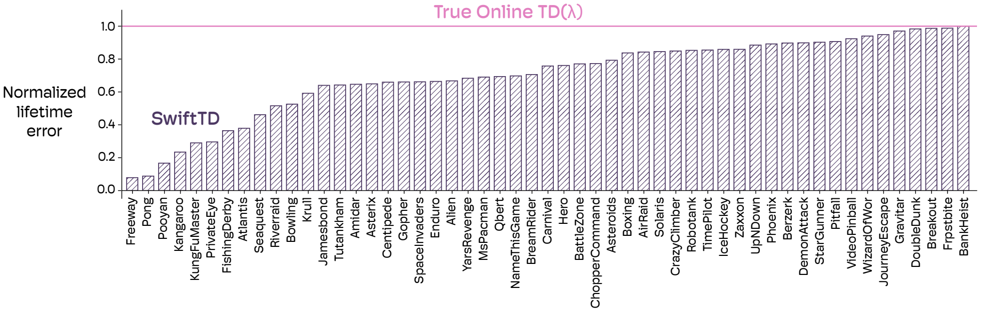

Figure 7: Relative lifetime error of SwiftTD to True Online TD(λ) on the Atari Prediction Benchmark. SwiftTD achieved lower lifetime error than True Online TD(λ) in almost all games.

We plot individual learning curves for eight games in Figure 6. In each plot the y-axis is the lifetime error and the x-axis is the lifetime. In each of the eight games in Figure 6, SwiftTD had a smaller lifetime error for almost all values of the lifetime parameter.

We visualize the predictions made by both methods in the final 3,000 steps on four games in Figure 1. The gray dotted lines are the return from each time step. Predictions learned by SwiftTD were significantly more accurate. In some games—Altantis, Pooyan—True Online TD(λ) completely failed for all hyperparameter settings whereas SwiftTD learned accurate predictions.

We also compared the performance of SwiftTD and True Online TD(λ) on all games. For better visualization, we divided the lifetime error of both methods by the lifetime error achieved by True Online TD(λ). After division, True Online TD(λ) had a normalized error of one. We visualize all errors in Figure 7. SwiftTD performed as well or better on all games. In some games, the error achieved by SwiftTD was an order of magnitude lower.

Overall, SwiftTD performed better than True Online TD(λ) on the Atari prediction benchmark and appears to satisfy our goals of fast and robust TD learning.

7 Combining SwiftTD with Neural Network

So far we have compared all methods with linear learners. In this section, we share one way SwiftTD can be combined with neural networks.

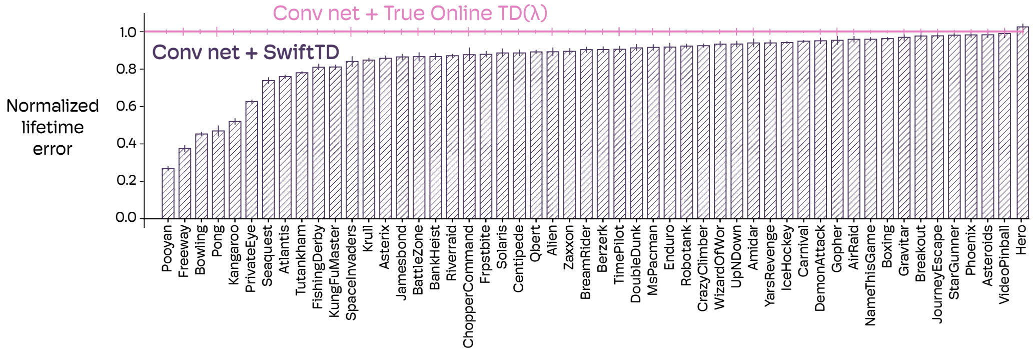

Figure 8: Comparing performance of convolutional networks on the Atari Prediction Benchmark. SwiftTD significantly outperformed True Online TD(λ) even when combined with neural networks. The confidence intervals are $+-$ two standard error around the mean computed over fifteen runs.

Instead of using the preprocessing described earlier, we used SwiftTD with a one layer convolutional neural network. We applied a convolutional layer on the 105 x 80 x 24 tensor we got after stacking the three tensors given by the binning process. The convolutional layer had 25 kernels of size 3 × 3 × 24 each. The weights of the kernels were initialized by sampling from $\mathcal{U}(-1, 1)$.

We applied all the kernels to the input tensor with a stride of 2. The output of the convolutional layer was a 52 x 40 x 25 tensor. We passed the output through the ReLU activation function (Fukushima, 1969) and flattened the tensor to get a vector with 52,000 components. The vector is linearly combined with a weight vector to make predictions.

The main challenge in applying SwiftTD to neural networks was that SwiftTD was developed for linear learners. We got past this limitation by applying SwiftTD to only the last layer of the network and updated the weights of the kernels using TD(λ), similar to Tesauro (1995). For our baseline, we used True Online TD(λ) in the last layer. We tuned the step-size parameter of weights in the kernels independently of the hyperparameters of the learners in the last layer.

The results of convolutional networks with SwiftTD and True Online TD(λ) are in Figure 8. Similar to the linear case, SwiftTD helped in almost all games. Results with convolutional neural networks highlight that simply using SwiftTD for the weights in the last layer of existing Deep-RL systems could improve their performance.

8 Conclusions and Future Work

SwiftTD has the potential to be the go-to algorithm for learning predictions from online streams of data; it unlocks the possibility of computationally efficient few-shot learning. The combination of SwiftTD with efficient learning algorithms for RNNs (Menick et al., 2021; Javed et al., 2023) is a particularly promising direction for replay-free state construction from an online stream of data.

Acknowledgements

We are grateful to the Alberta Machine Intelligence Institute (Amii) and NSERC for funding this research and to The Digital Research Alliance of Canada for providing computational resources. We are also thankful to the anonymous reviewers for improving the paper with useful feedback.

References

Bellemare, M. G., Naddaf, Y., Veness, J., & Bowling, M. (2013). The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research.*

Dabney, W., & Barto, A. (2012). Adaptive step-size for online temporal difference learning. *AAAI Conference on Artificial Intelligence.*

Degris, T., Javed, K., Sharifnassab, A., Liu, Y., & Sutton, R. S. (2024). Step-size optimization for continual learning. *arXiv preprint arXiv:2401.17401.*

Fukushima, K. (1969). Visual feature extraction by a multilayered network of analog threshold elements. *IEEE Transactions on Systems Science and Cybernetics.*

Fujita, Y., Nagarajan, P., Kataoka, T., & Ishikawa, T. (2021). ChainerRL: A deep reinforcement learning library. *Journal of Machine Learning Research.*

Ghiassian, S. (2022). *Online off-policy prediction.* [Doctoral dissertation, University of Alberta].

Hessel, M., Modayil, J., Van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., ... & Silver, D. (2018). Rainbow: Combining improvements in deep reinforcement learning. *AAAI Conference on Artificial Intelligence.*

Javed, K., White, M., & Sutton, R. S. (2021). Scalable online recurrent learning using columnar neural networks. *arXiv preprint arXiv:2103.05787.*

Javed, K., Shah, H., Sutton, R. S., & White, M. (2023). Scalable real-time recurrent learning using columnar-constructive networks. *Journal of Machine Learning Research.*

Javed, K. (2024). *Real-time Reinforcement Learning for Achieving Goals in Big Worlds.* [Doctoral dissertation, University of Alberta].

Javed, K., Sutton, R. S. (2024). The big world hypothesis and its ramifications for artificial intelligence. *Finding the Frame Workshop, Reinforcement Learning Conference.*

Kingma, D. P., & Ba, J. (2015). Adam: A method for stochastic optimization. *International Conference on Learning Representations.*

Kearney, A., Veeriah, V., Travnik, J. B., Sutton, R. S., & Pilarski, P. M. (2018). Tidbd: Adapting temporal-difference step-sizes through stochastic meta-descent. *arXiv preprint arXiv:1804.03334.*

Konda, V., & Tsitsiklis, J. (1999). Actor-critic algorithms. *Advances in Neural Information Processing Systems.*

Li, Z., Zhou, F., Chen, F., & Li, H. (2017). Meta-SGD: Learning to learn quickly for few-shot learning. *arXiv preprint arXiv:1707.09835.*

Mahmood, A. R., Sutton, R. S., Degris, T., & Pilarski, P. M. (2012). Tuning-free step-size adaptation. *IEEE International Conference on Acoustics, Speech and Signal processing.*

Menick, J., Elsen, E., Evci, U., Osindero, S., Simonyan, K., & Graves, A. (2021). A practical sparse approximation for real time recurrent learning. *International Conference on Learning Representations.*

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., ..., & Hassabis, D. (2015). Human-level control through deep reinforcement learning. *Nature.*

Rummery, A., & Niranjan, M. On-line Q-learning using connectionist systems (1994). *Department of Engineering, University of Cambridge.*

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347.*

Sutton, R. S. (1988). Learning to predict by the methods of temporal differences. *Machine Learning.*

Sutton, R. S., & Barto, A. G. (2018). *Reinforcement learning: An introduction.* MIT Press.

Sutton, R. S. (1992). Adapting bias by gradient descent: An incremental version of delta-bar-delta. *AAAI Conference on Artificial Intelligence.*

Sutton, R. S., Modayil, J., Delp, M., Degris, T., Pilarski, P. M., White, A., & Precup, D. (2011). Horde: A scalable real-time architecture for learning knowledge from unsupervised sensorimotor interaction. *International Conference on Autonomous Agents and Multiagent Systems.*

Tesauro, G. (1995). Temporal difference learning and TD-gammon. *Communications of the ACM.*

Thill, M. (2015). *Temporal difference learning methods with automatic step-size adaption for strategic board games: Connect-4 and Dots-and-Boxes.* [Masters dissertation, Cologne University of Applied Sciences].

Tieleman, T., & Hinton, G. (2012). *Lecture 6.5-rmsprop, Coursera: Neural networks for machine learning.* [Technical report, University of Toronto].

Van Hasselt, H., & Sutton, R. S. (2015). Learning to predict independent of span. *arXiv preprint arXiv:1508.04582.*

Van Seijen, H., Mahmood, A. R., Pilarski, P. M., Machado, M. C., & Sutton, R. S. (2016). True online temporal-difference learning. *Journal of Machine Learning Research.*

Watkins, C. J., & Dayan, P. (1992). Q-learning. *Machine Learning.*

Williams, R. J., & Zipser, D. (1989). A learning algorithm for continually running fully recurrent neural networks. *Neural Computation.*

Young, K., Wang, B., & Taylor, M. E. (2018). Metatrace actor-critic: Online step-size tuning by meta-gradient descent for reinforcement learning control. *arXiv preprint arXiv:1805.04514.*

A Step-size Optimization for TD Learning

In this section, we derive efficient and approximate algorithms for step-size optimization for TD(λ) and True Online TD(λ).

A.1 Step-size Optimization for TD(λ)

TD(λ) uses the λ-return as the target. The λ-return at time step t is defined as:

$$G_t^\lambda = (1-\lambda)\sum_{n=1}^{\infty} \lambda^{n-1} G_{t:t+\underset{n}{\cancel{1}}} \tag{9}$$

The step-size parameters are parametrized as a vector of values $\boldsymbol{\beta} \in \mathbb{R}^n$, where the ith step-size parameter is $e^{\boldsymbol{\beta}[i]}$. TD($\lambda$) updates the ith weight as:

$$\boldsymbol{w}_t[i] = \boldsymbol{w}_{t-1}[i] + \delta_t \boldsymbol{z}_{t-1}[i], \quad \text{基于资格迹的反向更新视角} \tag{10}$$

where $\boldsymbol{z}_t[i]$ is updated as:

$$\boldsymbol{z}_t[i] = \gamma\lambda \boldsymbol{z}_{t-1}[i] + e^{\boldsymbol{\beta}_t[i]}\boldsymbol{\phi}_t[i]. \quad \text{z 是资格迹} \tag{11}$$

The prediction at time step t can be made using the weight vector before or after the update (*i.e.*, using $\boldsymbol{w}_{t-1}$ or $\boldsymbol{w}_t$) and both are used in different parts of the update. To differentiate between the two, we define v_{t_1,t_2} as

$$v_{t_1,t_2} \overset{\text{def}}{=} \sum_{i=1}^{n} \boldsymbol{w}_{t_1}[i]\boldsymbol{\phi}_{t_2}[i]. \tag{12}$$

The objective for step-size optimization is to minimize the sum of squared error between the prediction and the λ-return. To update the step-size parameters, we can use the meta-gradient of the sum of squared error. The meta-gradient is:

Algorithm 2: TD(λ) with Step-size Optimization

```
Parameters: $\alpha, \lambda$
Initialize: $(\boldsymbol{w}, \boldsymbol{z}) \leftarrow (\mathbf{0}, \mathbf{0}) \in \mathbb{R}^n, (v^{old}, v^{\delta}) = (0, 0)$
while alive do
    Receive $\boldsymbol{\phi}$, $\gamma$, and $r$
    $v \leftarrow \sum_{\boldsymbol{\phi}[i] \neq 0} \boldsymbol{w}[i]\boldsymbol{\phi}[i]$
    $\delta \leftarrow r + \gamma v - v^{old}$
    for $z_i \neq 0$ do
        $\boldsymbol{\delta}^{\boldsymbol{w}}[i] \leftarrow \delta \boldsymbol{z}[i]$
        $\boldsymbol{w}[i] \leftarrow \boldsymbol{w}[i] + \boldsymbol{\delta}^{\boldsymbol{w}}[i]$
        $\boldsymbol{\beta}[i] \leftarrow \boldsymbol{\beta}[i] + \frac{\theta}{e^{\boldsymbol{\beta}[i]}+\epsilon}\delta \boldsymbol{p}[i]$
        $\boldsymbol{h}[i] \leftarrow \boldsymbol{h}^{\boldsymbol{temp}}[i]$
        $\boldsymbol{h}^{\boldsymbol{temp}}[i] \leftarrow \boldsymbol{h}[i] + \boldsymbol{z}[i]\delta$
        $(\boldsymbol{z}[i], \boldsymbol{p}[i]) \leftarrow (\gamma\lambda \boldsymbol{z}[i], \gamma\lambda \boldsymbol{p}[i])$
    for $\boldsymbol{\phi}[i] \neq 0$ do
        $\boldsymbol{z}[i] \leftarrow \boldsymbol{z}[i] + \alpha \boldsymbol{\phi}[i]$
        $\boldsymbol{p}[i] \leftarrow \boldsymbol{p}[i] + \boldsymbol{\phi}[i]\boldsymbol{h}[i]$
        $\boldsymbol{h}^{\boldsymbol{temp}}[i] \leftarrow \boldsymbol{h}^{\boldsymbol{temp}}[i] - \boldsymbol{h}[i]\boldsymbol{z}[i]\boldsymbol{\phi}[i]$
    $v^{old} \leftarrow \sum_{\boldsymbol{\phi}[i] \neq 0} \boldsymbol{w}[i]\boldsymbol{\phi}[i]$
```

$$\frac{\partial \mathcal{L}(t)}{\partial \boldsymbol{\beta}[i]} = \frac{\partial}{\partial \boldsymbol{\beta}[i]} \frac{(G_t^\lambda - v_{t-1,t})^2}{2} = (G_t^\lambda - v_{t-1,t}) \frac{\partial v_{t-1,t}}{\partial \boldsymbol{\beta}[i]} \tag{13}$$

$$= (G_t^\lambda - v_{t-1,t}) \frac{\partial}{\partial \boldsymbol{\beta}[i]} \sum_{j=1}^{n} \boldsymbol{w}_{t-1}[j] \boldsymbol{\phi}_t[j] \tag{14}$$

$$= (G_t^\lambda - v_{t-1,t}) \sum_{j=1}^{n} \boldsymbol{\phi}_t[j] \frac{\partial \boldsymbol{w}_{t-1}[j]}{\partial \boldsymbol{\beta}[i]}. \tag{15}$$

Similar to the author of IDBD, we assume that the indirect impact of $\boldsymbol{\beta}[i]$ on $\boldsymbol{w}[j]$ for $j \neq i$ is negligible. Intuitively, this approximation makes sense as changing $e^{\boldsymbol{\beta}[i]}$ will mostly impact $\boldsymbol{w}[i]$. For a more detailed discussion on this approximation, see Javed et al. (2021). Using this approximation we get:

$$(G_t^\lambda - v_{t-1,t}) \sum_{j=1}^{n} \boldsymbol{\phi}_t[j] \frac{\partial \boldsymbol{w}_{t-1}[j]}{\partial \boldsymbol{\beta}[i]} \approx (G_t^\lambda - v_{t-1,t}) \boldsymbol{\phi}_t[i] \frac{\partial \boldsymbol{w}_{t-1}[i]}{\partial \boldsymbol{\beta}[i]}, \tag{16}$$

where δ_t is

$$\delta_t \overset{\text{def}}{=} r_t + \gamma v_{t-1,t} - v_{t-1,t-1}. \tag{17}$$

We define $\frac{\partial \boldsymbol{w}_t[i]}{\partial \boldsymbol{\beta}[i]}$ as $\boldsymbol{h}_t[i]$. We can compute $\boldsymbol{h}_t[i]$ recursively as:

$$\begin{aligned} \boldsymbol{h}_t[i] &= \frac{\partial \boldsymbol{w}_t[i]}{\partial \boldsymbol{\beta}[i]} \\ &= \frac{\partial \boldsymbol{w}_{t-1}[i]}{\partial \boldsymbol{\beta}[i]} + \frac{\partial (\delta_t \boldsymbol{z}_{t-1}[i])}{\partial \boldsymbol{\beta}[i]} \\ &= \boldsymbol{h}_{t-1}[i] + \boldsymbol{z}_{t-1}[i] \frac{\partial \delta_t}{\partial \boldsymbol{\beta}[i]} + \delta_t \frac{\partial \boldsymbol{z}_{t-1}[i]}{\partial \boldsymbol{\beta}[i]}. \end{aligned} \tag{18}$$

The gradient $\frac{\partial \delta_t}{\partial \boldsymbol{\beta}[i]}$ can be computed using the same approximation as IDBD as:

$$\begin{aligned} \frac{\partial \delta_t}{\partial \boldsymbol{\beta}[i]} &= \frac{\partial (r_t + \gamma \sum_{j=1}^n \boldsymbol{w}_{t-1}[j] \boldsymbol{\phi}_t[j] - \sum_{j=1}^n \boldsymbol{w}_{t-1}[j] \boldsymbol{\phi}_{t-1}[j])}{\partial \boldsymbol{\beta}[i]} \\ &\approx -\boldsymbol{h}_{t-1}[i] \boldsymbol{\phi}_{t-1}[i]. \end{aligned} \tag{19}$$

Finally, we define $\bar{\boldsymbol{z}}_t[i]$ as $\frac{\partial \boldsymbol{z}_t[i]}{\partial \boldsymbol{\beta}[i]}$. Then:

$$\begin{aligned} \bar{\boldsymbol{z}}_t[i] &= \frac{\partial}{\partial \boldsymbol{\beta}[i]} \left(\gamma \lambda \boldsymbol{z}_{t-1}[i] + e^{\boldsymbol{\beta}_t[i]} \boldsymbol{\phi}_t[i] \right) \\ &= \gamma \lambda \bar{\boldsymbol{z}}_{t-1}[i] + e^{\boldsymbol{\beta}_t[i]} \boldsymbol{\phi}_t[i] \\ &= \boldsymbol{z}_t[i]. \end{aligned} \tag{20}$$

The final $\boldsymbol{h}_t[i]$ update is:

$$\begin{aligned} \boldsymbol{h}_t[i] &\approx \boldsymbol{h}_{t-1}[i] - \boldsymbol{z}_{t-1}[i] \boldsymbol{h}_{t-1}[i] \boldsymbol{\phi}_{t-1}[i] + \delta_t \boldsymbol{z}_{t-1}[i]. \\ &= \boldsymbol{h}_{t-1}[i] \left(1 - \boldsymbol{z}_{t-1}[i] \boldsymbol{\phi}_{t-1}[i]\right) + \delta_t \boldsymbol{z}_{t-1}[i]. \end{aligned} \tag{21}$$

From Equation 15, we see that we still need to compute $(G_t^\lambda - v_{t-1,t})$. This is not a problem as we can write it as sum of TD errors as:

$$G_t^\lambda - v_{t-1,t} = \sum_{j=t+1}^{\infty} (\gamma \lambda)^{j-t-1} \delta_j. \tag{22}$$

The final update for $\boldsymbol{\beta}[i]$ is

$$\begin{aligned}\frac{\partial \mathcal{L}(t)}{\boldsymbol{\beta}[i]} &= \left(\sum_{j=t+1}^{\infty} (\gamma\lambda)^{j-t-1}\delta_j\right) \boldsymbol{h}_{t-1}[i]\boldsymbol{\phi}_t[i] \\ &= \left(\sum_{j=t+1}^{\infty} (\gamma\lambda)^{j-t-1}\boldsymbol{h}_{t-1}[i]\boldsymbol{\phi}_t[i]\delta_j\right) \\ &= \boldsymbol{h}_{t-1}[i]\boldsymbol{\phi}_t[i]\delta_{t+1} + \gamma\lambda\ \boldsymbol{h}_{t-1}[i]\boldsymbol{\phi}_t[i]\delta_{t+2} + \gamma^2\lambda^2\boldsymbol{h}_{t-1}[i]\boldsymbol{\phi}_t[i]\delta_{t+3} + \dots .\end{aligned} \tag{23}$$

The update involves future terms but can easily be done online using eligibility traces by accumulating $\boldsymbol{h}_{t-1}[i]\boldsymbol{\phi}_t[i]$ in a trace decayed by $\lambda\gamma$ at each time step, and applying the update overtime, as shown in Algorithm 2.

A.2 Step-size Optimization for True Online TD(λ)

True Online TD(λ) is a more complex algorithm and the updates for meta-gradients are also more involved. We define ${\delta'}_t$ as

Algorithm 3: True Online TD(λ) with Step-size Optimization

```
Parameters: $\alpha_{init}, \lambda, \theta$
Initialize: $(\boldsymbol{w}, \boldsymbol{h^{old}}, \boldsymbol{h^{temp}}, \boldsymbol{z^{\delta}}, \boldsymbol{p}, \boldsymbol{h}, \boldsymbol{z}, \bar{\boldsymbol{z}}) \leftarrow (\boldsymbol{0}, \cdots, \boldsymbol{0}), (v^{\delta}, v^{old}) = (0, 0), \boldsymbol{\beta} \leftarrow \ln(\boldsymbol{\alpha}_{init}) \in \mathbb{R}^n$
while alive do
    Receive $\boldsymbol{\phi}$, $\gamma$, and $r$
    $v \leftarrow \sum_{\boldsymbol{\phi}[i]\neq 0} \boldsymbol{w}[i]\boldsymbol{\phi}[i]$
    $\delta' \leftarrow r + \gamma v - v^{old}$
    for $\boldsymbol{z}[i] \neq 0$ do
        $\boldsymbol{\delta^{w}}[i] \leftarrow \delta'\boldsymbol{z}[i] - \boldsymbol{z^{\delta}}[i]v^{\delta}$
        $\boldsymbol{w}[i] \leftarrow \boldsymbol{w}[i] + \boldsymbol{\delta^{w}}[i]$
        $\boldsymbol{\beta}[i] \leftarrow \boldsymbol{\beta}[i] + \frac{\theta}{e^{\boldsymbol{\beta}[i]}+\epsilon}\delta'\boldsymbol{p}[i]$
        $\boldsymbol{h^{old}}[i] \leftarrow \boldsymbol{h}[i]$
        $\boldsymbol{h}[i] \leftarrow \boldsymbol{h^{temp}}[i]$
        $\boldsymbol{h^{temp}}[i] \leftarrow \boldsymbol{h}[i] + \delta'\bar{\boldsymbol{z}}[i] - \boldsymbol{z^{\delta}}[i]v^{\delta}$
        $\boldsymbol{z^{\delta}}[i] = 0$
        $(\boldsymbol{z}[i], \boldsymbol{p}[i], \bar{\boldsymbol{z}}[i]) \leftarrow (\gamma\lambda\boldsymbol{z}[i], \gamma\lambda\boldsymbol{p}[i], \gamma\lambda\bar{\boldsymbol{z}}[i])$
    $v^{\delta} \leftarrow 0$
    $T \leftarrow \sum_{\boldsymbol{\phi}[i]\neq 0} \boldsymbol{z}[i]\boldsymbol{\phi}[i]$
    for $\boldsymbol{\phi}[i] \neq 0$ do
        $v^{\delta} \leftarrow v^{\delta} + \boldsymbol{\delta^{w}}[i]\boldsymbol{\phi}[i]$
        $\boldsymbol{z^{\delta}}[i] \leftarrow e^{\boldsymbol{\beta}[i]}\boldsymbol{\phi}[i]$
        $\boldsymbol{z}[i] \leftarrow \boldsymbol{z}[i] + \boldsymbol{z^{\delta}}[i](1 - T)$
        $\boldsymbol{p}[i] \leftarrow \boldsymbol{p}[i] + \boldsymbol{\phi}[i]\boldsymbol{h}[i]$
        $\bar{\boldsymbol{z}}[i] \leftarrow \bar{\boldsymbol{z}}[i] + \boldsymbol{z^{\delta}}[i]\,[1 - T - \boldsymbol{\phi}[i]\bar{\boldsymbol{z}}[i]]$
        $\boldsymbol{h^{temp}}[i] \leftarrow \boldsymbol{h^{temp}}[i] - \boldsymbol{h^{old}}[i]\boldsymbol{\phi}[i]\left(\boldsymbol{z}[i] - \boldsymbol{z^{\delta}}[i]\right) - \boldsymbol{h}[i]\boldsymbol{z^{\delta}}[i]\boldsymbol{\phi}[i]$
    $v^{old} \leftarrow v$
```

$${\delta'}_t \overset{\text{def}}{=} r_t + \gamma v_{t-1,t} - v_{t-2,t-1}. \tag{24}$$

The weight update for True Online TD(λ) (Van Seijen et al., 2016) is:

$$\boldsymbol{w}_t[i] = \boldsymbol{w}_{t-1}[i] + {\delta'}_t\boldsymbol{z}_{t-1}[i] - e^{\boldsymbol{\beta}_{t-1}[i]}(v_{t-1,t-1} - v_{t-2,t-1})\boldsymbol{\phi}_{t-1}[i], \tag{25}$$

where $\boldsymbol{z}_{t-2}[i]$ is updated as:

$$\boldsymbol{z}_t[i] = \gamma\lambda\boldsymbol{z}_{t-1}[i] + e^{\boldsymbol{\beta}_t[i]}\boldsymbol{\phi}_t[i] - e^{\boldsymbol{\beta}_t[i]}\boldsymbol{\phi}_t[i]T_t. \tag{26}$$

The term T_t is defined as:

$$T_t \overset{\text{def}}{=} \gamma\lambda\sum_{i=1}^{n}\boldsymbol{z}_{t-1}[i]\boldsymbol{\phi}_t[i]. \tag{27}$$

We define $\frac{\partial \boldsymbol{w}_t[i]}{\partial \boldsymbol{\beta}[i]}$ as $\boldsymbol{h}_t[i]$. Then, we can expand $\boldsymbol{h}_t[i]$ recursively as:

$$\begin{aligned}
\boldsymbol{h}_t[i] &= \frac{\partial \boldsymbol{w}_t[i]}{\partial \boldsymbol{\beta}[i]} \\
&= \frac{\partial \boldsymbol{w}_{t-1}[i]}{\partial \boldsymbol{\beta}[i]} + \frac{\delta(\delta'_t\boldsymbol{z}_{t-1}[i])}{\partial \boldsymbol{\beta}[i]} - \boldsymbol{\phi}_{t-1}[i]\frac{\partial\left(e^{\boldsymbol{\beta}_{t-1}[i]}(v_{t-1,t-1} - v_{t-2,t-1})\right)}{\partial \boldsymbol{\beta}[i]} \\
&= \boldsymbol{h}_{t-1}[i] + \boldsymbol{z}_{t-1}[i]\frac{\partial \delta'_t}{\partial \boldsymbol{\beta}[i]} + \delta'_t\frac{\partial \boldsymbol{z}_{t-1}[i]}{\partial \boldsymbol{\beta}[i]} - \boldsymbol{\phi}_{t-1}[i]e^{\boldsymbol{\beta}_{t-1}[i]}\frac{\partial(v_{t-1,t-1} - v_{t-2,t-1})}{\partial \boldsymbol{\beta}[i]} \\
&- \boldsymbol{\phi}_{t-1}[i](v_{t-1,t-1} - v_{t-2,t-1})e^{\boldsymbol{\beta}_{t-1}[i]}.
\end{aligned} \tag{28}$$

Using the IDBD approximation again, we can simplify the gradient as:

$$\begin{aligned}
\boldsymbol{h}_t[i] \approx \boldsymbol{h}_{t-1}[i] + \boldsymbol{z}_{t-1}[i]\frac{\partial(\delta'_t)}{\partial \boldsymbol{\beta}[i]} + \delta'_t\frac{\partial \boldsymbol{z}_{t-1}[i]}{\partial \boldsymbol{\beta}[i]} - \boldsymbol{\phi}_{t-1}[i]e^{\boldsymbol{\beta}_{t-1}[i]}(\boldsymbol{h}_{t-1}[i]\boldsymbol{\phi}_{t-1}[i] - \boldsymbol{h}_{t-2}[i]\boldsymbol{\phi}_{t-1}[i]) \\
- \boldsymbol{\phi}_{t-1}[i](v_{t-1,t-1} - v_{t-2,t-1})e^{\boldsymbol{\beta}_{t-1}[i]}.
\end{aligned} \tag{29}$$

The gradient $\frac{\partial \delta'_t}{\partial \boldsymbol{\beta}[i]}$ can be computed using the same approximation as IDBD as:

$$\begin{aligned}
\frac{\partial \delta'_t}{\partial \boldsymbol{\beta}[i]} &= \frac{\partial(r_t + \gamma\sum_{j=1}^{n}\boldsymbol{w}_{t-1}[j]\boldsymbol{\phi}_t[j] - \sum_{j=1}^{n}\boldsymbol{w}_{t-2}[j]\boldsymbol{\phi}_{t-1}[j])}{\partial \boldsymbol{\beta}[i]} \\
&\approx -\boldsymbol{h}_{t-2}[i]\boldsymbol{\phi}_{t-1}[i].
\end{aligned} \tag{30}$$

Finally, let us define $\bar{\boldsymbol{z}}_t[i]$ as $\frac{\partial \boldsymbol{z}_t[i]}{\partial \boldsymbol{\beta}[i]}$. Then:

$$\begin{aligned}
\bar{\boldsymbol{z}}_t[i] &= \frac{\partial}{\partial \boldsymbol{\beta}[i]}\left(\gamma\lambda e_{t-1} + e^{\boldsymbol{\beta}_t[i]}\boldsymbol{\phi}_t[i] - e^{\boldsymbol{\beta}_t[i]}\boldsymbol{\phi}_t[i]T_t\right) \\
&= \gamma\lambda\bar{\boldsymbol{z}}_{t-1}[i] + e^{\boldsymbol{\beta}_t[i]}\boldsymbol{\phi}_t[i] - e^{\boldsymbol{\beta}_t[i]}\boldsymbol{\phi}_t[i]T_t - e^{\boldsymbol{\beta}_t[i]}\boldsymbol{\phi}_t[i]\frac{\partial T_t}{\partial \boldsymbol{\beta}[i]} \\
&\approx \gamma\lambda\bar{\boldsymbol{z}}_{t-1}[i] + e^{\boldsymbol{\beta}_t[i]}\boldsymbol{\phi}_t[i] - e^{\boldsymbol{\beta}_t[i]}\boldsymbol{\phi}_t[i]T_t - \gamma\lambda e^{\boldsymbol{\beta}_t[i]}\boldsymbol{\phi}_t[i]^2\bar{\boldsymbol{z}}_{t-1}[i] \\
&= \gamma\lambda\bar{\boldsymbol{z}}_{t-1}[i] + e^{\boldsymbol{\beta}_t[i]}\boldsymbol{\phi}_t[i]\left(1 - T_t - \gamma\lambda\boldsymbol{\phi}_t[i]\bar{\boldsymbol{z}}_{t-1}[i]\right).
\end{aligned} \tag{31}$$

The final $\boldsymbol{h}_t[i]$ update is:

$$\begin{aligned}
\boldsymbol{h}_t[i] &\approx \boldsymbol{h}_{t-1}[i] - \boldsymbol{z}_{t-1}[i]\boldsymbol{h}_{t-2}[i]\boldsymbol{\phi}_{t-1}[i] \\
&+ \delta'_t\bar{\boldsymbol{z}}_{t-1}[i] - \boldsymbol{\phi}_{t-1}[i]e^{\boldsymbol{\beta}_{t-1}[i]}(\boldsymbol{h}_{t-1}[i]\boldsymbol{\phi}_{t-1}[i] - \boldsymbol{h}_{t-2}[i]\boldsymbol{\phi}_{t-1}[i]) \\
&- \boldsymbol{\phi}_{t-1}[i](v_{t-1,t-1} - v_{t-2,t-1})e^{\boldsymbol{\beta}_{t-1}[i]}. \\
&= \boldsymbol{h}_{t-1}[i] - \boldsymbol{h}_{t-2}[i]\boldsymbol{\phi}_{t-1}[i]\left(\boldsymbol{z}_{t-1}[i] - e^{\boldsymbol{\beta}_{t-1}[i]}\boldsymbol{\phi}_{t-1}[i]\right) \\
&+ \delta'_t\bar{\boldsymbol{z}}_{t-1}[i] - \boldsymbol{\phi}_{t-1}[i]e^{\boldsymbol{\beta}_{t-1}[i]}\boldsymbol{h}_{t-1}[i]\boldsymbol{\phi}_{t-1}[i] \\
&- \boldsymbol{\phi}_{t-1}[i](v_{t-1,t-1} - v_{t-2,t-1})e^{\boldsymbol{\beta}_{t-1}[i]}.
\end{aligned} \tag{32}$$

The rest of the derivation is the same as TD(λ) with step-size optimization. The pseudocode for True Online TD(λ) with step-size optimization is given in Algorithm 3.

Algorithm 4: TD(λ) with the Overshoot Bound

```
Parameters: $\alpha, \lambda$
Initialize: $\boldsymbol{w} \leftarrow \mathbf{0} \in \mathbb{R}^n, \boldsymbol{z} \leftarrow \mathbf{0} \in \mathbb{R}^n, (v^{old} = 0$
while alive do
    Receive $\boldsymbol{\phi}$, $\gamma$, and $r$
    $v \leftarrow \sum_{\boldsymbol{\phi}[i] \neq 0} \boldsymbol{w}[i]\boldsymbol{\phi}[i]$
    $\delta \leftarrow r + \gamma v - v^{old}$
    for $z_i \neq 0$ do
        $\boldsymbol{w}[i] \leftarrow \boldsymbol{w}[i] + \delta \boldsymbol{z}[i]$; $\boldsymbol{z}[i] \leftarrow \gamma\lambda\boldsymbol{z}[i]$;
    $\tau \leftarrow \sum_{\boldsymbol{\phi}[i] \neq 0} \alpha\boldsymbol{\phi}[i]^2$
    for $\phi_i \neq 0$ do
        $\boldsymbol{z}[i] \leftarrow \boldsymbol{z}[i] + \min(1, \frac{1}{\tau})\alpha\boldsymbol{\phi}[i]$
    $v^{old} \leftarrow \sum_{\boldsymbol{\phi}[i] \neq 0} \boldsymbol{w}[i]\boldsymbol{\phi}[i]$
```

B Derivation of the Overshoot Bound

If we know the update rule for the parameters, we can derive a simpler expression for τ. Consider the case of linear regression when the weights are updated using stochastic gradient descent and the gradient is computed for the squared loss *i.e.*,

$$\frac{1}{2}\left(y_t^* - \sum_{i=0}^{n} \boldsymbol{w}_{t-1}[i]\boldsymbol{\phi}_t[i]\right)^2. \tag{33}$$

The weight update for the ith component of the parameter vector is:

$$\boldsymbol{w}_t[i] = \boldsymbol{w}_{t-1}[i] + \boldsymbol{\alpha}[i]\boldsymbol{\phi}_t[i]\delta_t, \tag{34}$$

where δ_t is the prediction error. The correction ratio can be simplified as:

$$\begin{aligned}
\tau &= \frac{(y_t^* - \sum_{i=1}^{n} \boldsymbol{w}_t[i]\boldsymbol{\phi}_t[i]) - (y_t^* - \sum_{i=1}^{n} \boldsymbol{w}_{t+1}[i]\boldsymbol{\phi}_t[i])}{(y_t^* - \sum_{i=1}^{n} \boldsymbol{w}_t[i]\boldsymbol{\phi}_t[i])} \\
\implies \tau\left(y_t^* - \sum_{i=1}^{n} \boldsymbol{w}_t[i]\boldsymbol{\phi}_t[i]\right) &= (y_t^* - \sum_{i=1}^{n} \boldsymbol{w}_t[i]\boldsymbol{\phi}_t[i]) - (y_t^* - \sum_{i=1}^{n} \boldsymbol{w}_{t+1}[i]\boldsymbol{\phi}_t[i]) \\
\tau\left(y_t^* - \sum_{i=1}^{n} \boldsymbol{w}_t[i]\boldsymbol{\phi}_t[i]\right) &= -\sum_{i=1}^{n} \boldsymbol{w}_t[i]\boldsymbol{\phi}_t[i] + \sum_{i=1}^{n}(\boldsymbol{w}_t[i] + \alpha\delta\boldsymbol{\phi}[i])\boldsymbol{\phi}_t[i] \\
\tau\left(y_t^* - \sum_{i=1}^{n} \boldsymbol{w}_t[i]\boldsymbol{\phi}_t[i]\right) &= \sum_{i=1}^{n} \alpha\delta\boldsymbol{\phi}[i]^2 \\
\tau\left(y_t^* - \sum_{i=1}^{n} \boldsymbol{w}_t[i]\boldsymbol{\phi}_t[i]\right) &= \delta\sum_{i=1}^{n} \alpha\boldsymbol{\phi}[i]^2 \\
\tau\left(y_t^* - \sum_{i=1}^{n} \boldsymbol{w}_t[i]\boldsymbol{\phi}_t[i]\right) &= (y_t^* - \sum_{i=1}^{n} \boldsymbol{w}_t[i]\boldsymbol{\phi}_t[i])\sum_{i=1}^{n} \alpha\boldsymbol{\phi}[i]^2 \\
\tau &= \sum_{i=1}^{n} \alpha\boldsymbol{\phi}[i]^2.
\end{aligned} \tag{35}$$

B.1 Bound on the Correction Ratio for Stability

A general and sensible inductive bias in a learning mechanism is that an update to the parameters of an agent on a given sample should not worsen the prediction on the same sample. If the target for a given sample is y and the agent's prediction is $0.5y$, then after the update the prediction on the same sample should be closer to y than before—between $0.5y$ and y. [2]

This inductive bias can be implemented as a constraint on the update rule. The loss for our linear learner is:

$$\frac{1}{2}\left(y^*_t - \sum_{i=1}^{n} \boldsymbol{w}_{t-1}[i]\boldsymbol{\phi}_t[i]\right)^2. \tag{36}$$

The weight update for the ith parameter is

$$\boldsymbol{w}_t[i] = \boldsymbol{w}_{t-1}[i] + \alpha\boldsymbol{\phi}_t[i]\delta_t. \tag{37}$$

We can measure the change in error after the update on a sample as:

$$\left(y^*_t - \sum_{i=1}^{n} \boldsymbol{w}_{t-1}[i]\boldsymbol{\phi}_t[i]\right)^2 - \left(y^*_t - \sum_{i=1}^{n} \boldsymbol{w}_{t}[i]\boldsymbol{\phi}_t[i]\right)^2. \tag{38}$$

This quantity should be positive to minimize the error. This is equivalent to:

$$\left(y^*_t - \sum_{i=1}^{n} \boldsymbol{w}_{t}[i]\boldsymbol{\phi}_t[i]\right)^2 - \left(y^*_t - \sum_{i=1}^{n} \boldsymbol{w}_{t+1}[i]\boldsymbol{\phi}_t[i]\right)^2 > 0$$

$$\begin{aligned}
\text{Case 1: } y^*_t - \sum_{i=1}^{n} \boldsymbol{w}_t[i]\boldsymbol{\phi}_t[i] \geq 0 \text{ and } y^*_t - \sum_{i=1}^{n} \boldsymbol{w}_{t+1}[i]\boldsymbol{\phi}_t[i] \geq 0 \\
\implies y^*_t - \sum_{i=1}^{n} \boldsymbol{w}_t[i]\boldsymbol{\phi}_t[i] > y^*_t - \sum_{i=1}^{n} \boldsymbol{w}_{t+1}[i]\boldsymbol{\phi}_t[i] \geq 0 \\
\sum_{i=1}^{n} \boldsymbol{w}_t[i]\boldsymbol{\phi}_t[i] < \sum_{i=1}^{n} \boldsymbol{w}_{t+1}[i]\boldsymbol{\phi}_t[i] \leq y \\
\sum_{i=1}^{n} \boldsymbol{w}_t[i]\boldsymbol{\phi}_t[i] < \sum_{i=1}^{n} \left(\boldsymbol{w}_t[i] + \alpha\boldsymbol{\phi}_t[i]\delta\right)\boldsymbol{\phi}_t[i] \leq y \\
0 < \sum_{i=1}^{n} \left(\alpha\boldsymbol{\phi}_t[i]^2\delta\right) \leq y - \sum_{i=1}^{n} \boldsymbol{w}_t[i]\boldsymbol{\phi}_t[i] \\
0 < \delta_t \sum_{i=1}^{n} \alpha\boldsymbol{\phi}_t[i]^2 \leq \delta_t \\
0 < \sum_{i=1}^{n} \alpha\boldsymbol{\phi}_t[i]^2 \leq 1.
\end{aligned} \tag{39}$$

$$\begin{aligned}
\text{Case 2: } y^*_t - \sum_{i=1}^{n} \boldsymbol{w}_t[i]\boldsymbol{\phi}_t[i] < 0 \text{ and } y^*_t - \sum_{i=1}^{n} \boldsymbol{w}_{t+1}[i]\boldsymbol{\phi}_t[i] < 0 \\
\implies 0 < -y^*_t + \sum_{i=1}^{n} \boldsymbol{w}_t[i]\boldsymbol{\phi}_t[i] \geq -y^*_t + \sum_{i=1}^{n} \boldsymbol{w}_{t+1}[i]\boldsymbol{\phi}_t[i] \\
\implies 1 \leq \sum_{i=1}^{n} \alpha\boldsymbol{\phi}_t[i]^2 < 0. \\
\implies 1 \leq \tau_t < 0.
\end{aligned} \tag{40}$$

When $y^*_t - \sum_{i=1}^{n} \boldsymbol{w}_t[i]\boldsymbol{\phi}_t[i] > 0$ and $y^*_t - \sum_{i=1}^{n} \boldsymbol{w}_{t+1}[i]\boldsymbol{\phi}_t[i] < 0$ (or vice-versa) implies that the learner would overshoot the target after the update. We ignore that case because we want the prediction after the update to never overshoot the target.

[2] A more general constraint is that the prediction after the update should be in the between $0.5y$ and $1.5y$, but we limit ourselves to never overshooting in this work.

Algorithm 5: Linear Regression with the Learning-rate Bound

```
Parameters: $\boldsymbol{\alpha}$
Initialize: $\boldsymbol{w} \leftarrow \boldsymbol{0} \in \mathbb{R}^n$
while alive do
    Receive $\boldsymbol{\phi}$, and $y^*$
    $\delta \leftarrow y^* - \sum_{\boldsymbol{\phi}[i] \neq 0} \boldsymbol{w}[i]\boldsymbol{\phi}[i]$
    $\tau \leftarrow \sum_{i=1}^{n} \boldsymbol{\alpha}[i]\boldsymbol{\phi}[i]^2$
    for $\boldsymbol{\phi}[i] \neq 0$ do
        $\boldsymbol{w}[i] \leftarrow \boldsymbol{w}[i] + \min(1, \frac{1}{\tau})\boldsymbol{\alpha}[i]\delta\boldsymbol{\phi}[i]$;
```

Algorithm 6: True Online TD(λ) with the Overshoot Bound

```
Parameters: $\eta = 0.5, \alpha^{init} = 10^{-7}, \lambda, \gamma, \theta$
Initialize: $\boldsymbol{w}, \boldsymbol{z^\delta}, \boldsymbol{z} \leftarrow \boldsymbol{0} \in \mathbb{R}^n; (v^\delta, v^{old}) = (0, 0)$
while alive do
    Receive $\boldsymbol{\phi}$, $\gamma$, and $r$
    $v \leftarrow \sum_{\boldsymbol{\phi}[i] \neq 0} \boldsymbol{w}[i]\boldsymbol{\phi}[i]$
    $\delta' \leftarrow r + \gamma v - v^{old}$
    for $\boldsymbol{z}[i] \neq 0$ do
        $\boldsymbol{\delta^w}[i] \leftarrow \delta'\boldsymbol{z}[i] - \boldsymbol{z^\delta}[i]v^\delta$
        $\boldsymbol{w}[i] \leftarrow \boldsymbol{w}[i] + \boldsymbol{\delta^w}[i]\ \boldsymbol{z^\delta}[i] = 0$
        $\boldsymbol{z}[i] \leftarrow \gamma\lambda\boldsymbol{z}[i]$
    $v^\delta \leftarrow 0$
    $\tau \leftarrow \sum_{\boldsymbol{\phi}[i] \neq 0} \alpha\boldsymbol{\phi}[i]^2$
    $T \leftarrow \sum_{\boldsymbol{\phi}[i] \neq 0} \boldsymbol{z}[i]\boldsymbol{\phi}[i]$
    for $\boldsymbol{\phi}[i] \neq 0$ do
        $v^\delta \leftarrow v^\delta + \boldsymbol{\delta^w}[i]\boldsymbol{\phi}[i]$
        $\boldsymbol{z^\delta}[i] \leftarrow \min(1, \frac{1}{\tau})\alpha\boldsymbol{\phi}[i]$
        $\boldsymbol{z}[i] \leftarrow \boldsymbol{z}[i] + \boldsymbol{z^\delta}[i](1 - T)$
    $v^{old} \leftarrow v$
```

Forcing $0 < \tau \leq 1$ for every update guarantees that all update reduce the prediction error and the prediction after the update does not overshoot the target. This bound can easily be implemented for linear regression by setting step-size of the update to be $\min(\frac{\alpha}{\sum_{i=1}^{n} \alpha_t \boldsymbol{\phi}_t[i]^2}, \alpha_t)$ at every time step as shown in Algorithm 5. AutoStep (Mahmood, 2012) used a similar bound to make linear regression robust. Our contribution is extending this bound to TD learning.

Van Siejen et al. (2016) and Van Hasselt and Sutton (2015) showed that if we scale the increment to eligibility vector by the step-size parameter and omit the step-size parameter in the weight update, we end up with an algorithm that has different step-size parameter for different updates for the λ-return target. Their analysis works for TD learning with per-feature step-size parameters as well.

Their analysis provides a way of implementing the overshoot bound for TD learning. We can limit the update to the eligibility vector similar to the way we limited the update to the weights in Algorithm 5.

Algorithm 7: True Online TD(λ) with the Overshoot Bound and Step-size Decay

```
Parameters: $\eta = 0.5, \alpha^{init} = 10^{-7}, \epsilon = 0.99, \lambda, \gamma, \theta$
Initialize: $\boldsymbol{w}, \boldsymbol{z^\delta}, \boldsymbol{z} \leftarrow \boldsymbol{0} \in \mathbb{R}^n; (v^\delta, v^{old}) = (0, 0)$
while alive do
    Receive $\boldsymbol{\phi}$, $\gamma$, and $r$
    $v \leftarrow \sum_{\boldsymbol{\phi}[i] \neq 0} \boldsymbol{w}[i]\boldsymbol{\phi}[i]$
    $\delta' \leftarrow r + \gamma v - v^{old}$
    for $\boldsymbol{z}[i] \neq 0$ do
        $\boldsymbol{\delta^w}[i] \leftarrow \delta' \boldsymbol{z}[i] - \boldsymbol{z^\delta}[i] v^\delta$
        $\boldsymbol{w}[i] \leftarrow \boldsymbol{w}[i] + \boldsymbol{\delta^w}[i]$ $\boldsymbol{z^\delta}[i] = 0$
        $\boldsymbol{z}[i] \leftarrow \gamma\lambda\boldsymbol{z}[i]$
    $v^\delta \leftarrow 0$
    $\tau \leftarrow \sum_{i=0}^{n} \boldsymbol{\alpha}[i]\boldsymbol{\phi}[i]^2$
    $T \leftarrow \sum_{\boldsymbol{\phi}[i] \neq 0} \boldsymbol{z}[i]\boldsymbol{\phi}[i]$
    for $\boldsymbol{\phi}[i] \neq 0$ do
        $v^\delta \leftarrow v^\delta + \boldsymbol{\delta^w}[i]\boldsymbol{\phi}[i]$
        $\boldsymbol{z^\delta}[i] \leftarrow \min(1, \frac{1}{\tau})\alpha\boldsymbol{\phi}[i]$
        $\boldsymbol{z}[i] \leftarrow \boldsymbol{z}[i] + \boldsymbol{z^\delta}[i](1 - T)$
        if $\tau > 0$ then
            $\boldsymbol{\alpha}[i] \leftarrow \boldsymbol{\alpha}[i]\epsilon^{\boldsymbol{\phi}[i]^2}$
    $v^{old} \leftarrow v$
```

C Credit Assignment to Features by SwiftTD

We visualize how much credit SwiftTD assigned to different pixel locations in different games. To do so, we define a quantity that captures lifetime credit received by the ith feature as:

$$\text{Credit}_i^T = \sum_{t=1}^{T} e^{\boldsymbol{\beta}_t[i]} \boldsymbol{\phi}_t[i]^2, \tag{41}$$

where $\boldsymbol{\phi}_t[i]$ and $\boldsymbol{\beta}_t[i]$ are the ith feature and step-size parameter at time t, respectively. Credit_i^T measures credit the ith feature was willing to accept.

We run an experiment with SwiftTD where the initial value of the step-size parameters is very small (10^{-8}) and all the credit received by features is because the step-size optimization increase their step-size parameters.

For visualizing, we remove the components of the $\textbf{Credit}^T$ vector associated with features for previous action and reward. After removing them, we get a vector with 201,600 components. We reshape this vector to a 105 x 80 x 24 tensor and sum over the third dimension to get a 105 x 80 matrix. Finally, we visualize this matrix in Figure 9. SwiftTD assigned credit to meaningful aspects of the game that are predictive of rewards and returns. For example, in Pong, it assigned credit to trajectories of the ball. In MsPacman, it assigned credit to the dots and the enemies.

D Hyperparameter Sweeps

For both SwiftTD and True Online TD(λ), we swept over the hyperparameters as shown in Table 1. We use the same hyperparameters for both the linear function approximation and the neural network experiments. The experiments with LFA are completely deterministic and do not require multiple runs. The experiment results with convolutional networks are stochastic due to the random

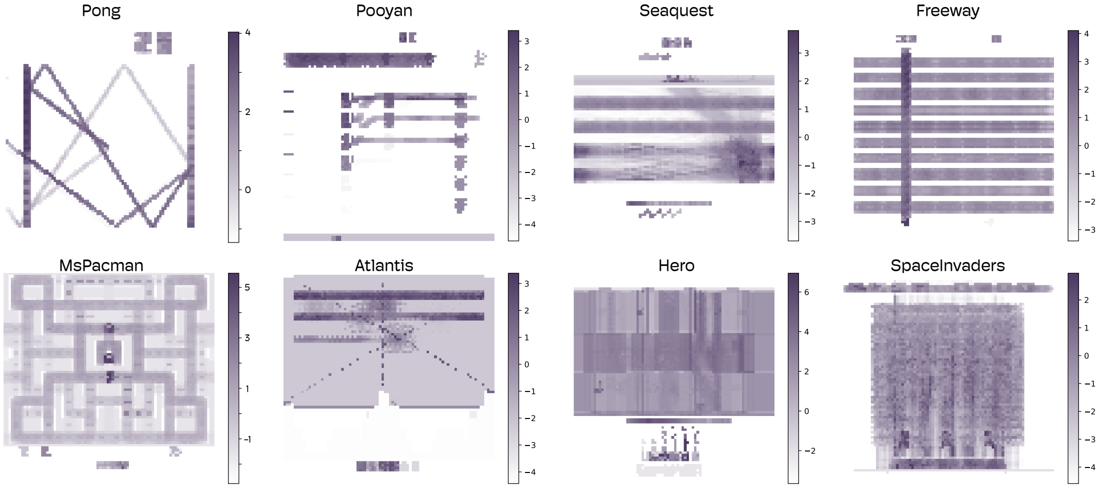

Figure 9: Visualizing the amount of credit assigned to each pixel by SwiftTD over the lifetime of the agent. The color map is in log space. We see that SwiftTD assigned credit to meaningful aspects of the game. For example, in Pong, it assigned credit to trajectories of the ball. In MsPacman, it assigned credit to the dots and the enemies. In SpaceInvaders, it assigned credit to the locations of enemies, bullets, and the UFO that passes at the top.

initialization of the weights. For statistical significance, we do a hyperparameter sweep with 5 runs for each configuration. We then find the best configuration, and do an additional 15 runs to report the results.

Symbol	Description	Algorithm	Values
$\boldsymbol{e^{\beta}}$	Step-size vector	SwiftTD	**0.0001**, **0.00001**
e^{β}	Step-size scalar	True Online TD(λ)	$3e^{-1}, 1e^{-1}, 3e^{-2}, 1e^{-2},$ $3e^{-3}, 1e^{-3}, 3e^{-4}, 1e^{-4}$ $3e^{-5}, 1e^{-5}, 3e^{-6}, 1e^{-6}$
α_{nn}	Scalar step-size of the kernels	Both	$1e^{-1}, 1e^{-2}, 1e^{-3}, 1e^{-4},$
λ	Compound return parameter	SwiftTD	0.95, 0.90, 0.80, 0.50, 0.0
λ	Compound return parameter	True Online TD(λ)	0.95, 0.90, 0.80, 0.50, 0.0
θ	Meta step-size	SwiftTD	$1e^{-2}, 1e^{-3}, 1e^{-4}$
η	Max correction ratio	SwiftTD	1.0, 0.5
ϵ	Decay factor	SwiftTD	0.9, 0.8

Table 1: Hyper-parameters used in the experiments. Note that the number of configurations for SwiftTD and True Online TD(λ) are the same. This is achieved by doing a much more fine-grained search for the step-size parameter of True Online TD(λ).

E Step-size Optimization With and Without the Semi-gradient Assumption

We used the semi-gradient assumption when deriving the meta-gradient for step-size optimization for both TD(λ) and True Online TD(λ). Intuitively, the semi-gradient version of the algorithm makes sense as both the step-size and the parameters are being optimized towards the same objective. However, one can argue that while semi-gradient makes sense for updating the parameters, it's better to optimize the step-size using the full-gradient. Conveniently, both versions can be implemented using traces and as a sanity check we compare the performance of TD(λ) with semi-gradient step-size

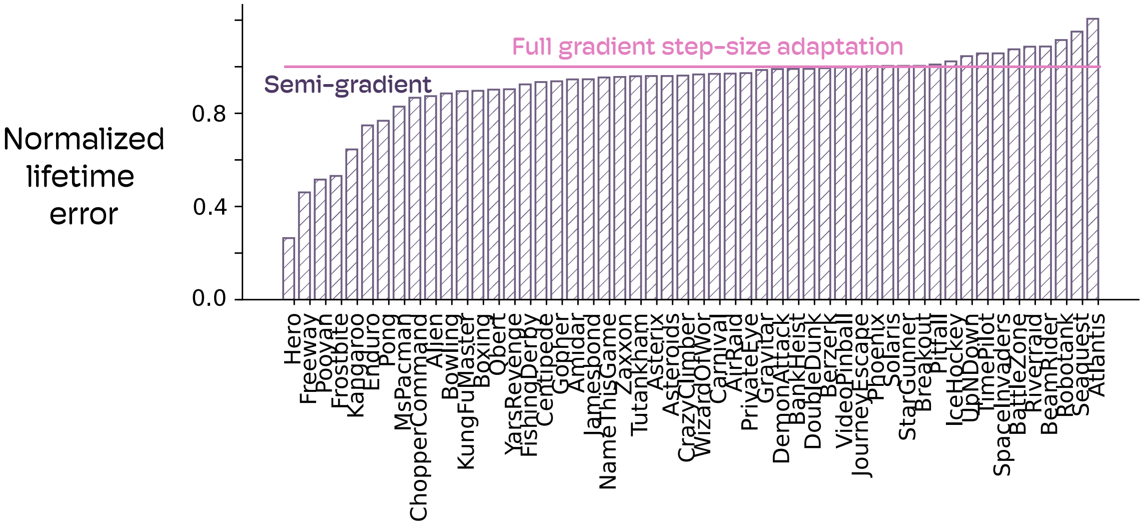

Figure 10: We compared TD(λ) with semi-gradient step-size optimization and TD(λ) with full-gradient step-size optimization. On average, semi-gradient performed better. In some environments, semi-gradient achieved less than half the error of full-gradient whereas even in the worst case of Atlantis, full gradient was only 20% better than semi-gradient.

optimization and TD(λ) with full-gradient step-size optimization in Figure 10. The results support the claim that the semi-gradient version is the better choice.