Real-time Reinforcement Learning for Achieving Goals in Big Worlds

by

Khurram Javed

A thesis submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Department of Computing Science

University of Alberta

Abstract

In this dissertation, I motivate the need for real-time learning and propose algorithms that can learn in real time. I argue that such algorithms are needed for achieving goals in large and partially observable environments—big worlds. I then present my algorithms, developed in collaboration with others, in two parts.

In Part I, I present algorithms that can learn quickly and reliably in the linear function approximation setting. I introduce an algorithm for learning temporal predictions—SwiftTD—and use it to develop an algorithm for decision-making—SwiftSarsa. The key property of these algorithms is that they can learn with large step-size parameters online without the instability associated with quick online learning.

In Part II, I present algorithms for learning non-linear recurrent features efficiently. I introduce the idea of continual imprinting for generating useful candidate features, and I present an algorithm for efficiently computing the gradients of recurrent features online.

Preface

Chapter 5, 6, and 7 are based on the paper:

- Javed, K., Sharifnassab, A., and Sutton, R. S. (2024). SwiftTD: Fast and Robust Temporal Difference Learning. *Reinforcement Learning Journal*,

which won the outstanding paper award at the reinforcement learning conference. Chapter 4 is based on a public seminar and a workshop paper. The workshop paper is:

- Javed, K., and Sutton, R. S. (2023). The Big World Hypothesis and its Ramifications for Artificial Intelligence. *Finding the Frame Workshop, RLC 2024*,

which was selected for an oral presentation, and the public seminar is:

- The Big World Hypothesis and its Ramifications, AI Seminar, University of Alberta, March 2023 (link).

Chapter 10 is based on the paper:

- Javed, K., Shah, H., Sutton, R. S. , White, M. (2023). Scalable Real-Time Recurrent Learning Using Columnar-Constructive Networks. *Journal of Machine Learning Research*,

which was published in JMLR and presented at ICML 2024 at its journal to conference track. Chapter 8 is based on work that is in progress, and Chapter 9 is based on two public talks that are:

- Real-time Online Learning by Imprinting at the Time of Low Plasticity, *AI Seminar, University of Alberta, October 2023*, and
- Real-time Reinforcement Learning using Dynamic Networks, *Cohere for AI, May 2024* (link).

To Mom and Dad,

for enabling me to pursue my goals and supporting my eccentric decisions.

Acknowledgements

The work presented in this dissertation started in weekly meetings with Rich. I was interested in online continual learning, and Rich was happy to spend time helping me refine my ideas. At some point our discussions turned into a PhD proposal.

The three years of my PhD were the most rewarding of my life. Early on I was frustrated by the poor review process of AI conferences. I proposed that we focus on developing new algorithms without being concerned about papers, and Rich supported this proposal. The decision to not focus on papers made it possible to explore a new frontier of learning algorithms.

Throughout my PhD, Rich was available to discuss, brainstorm, and provide feedback. On many occasions, his lines of inquiry led to insights and saved me months of work. In addition to shaping my research ideas, he also taught me how to think clearly and communicate effectively. I am grateful for his mentorship, wisdom, and friendship.

My committee members, Martha White, Joseph Modayil, Alona Fyshe, and Benjamin Van Roy, pushed me to think about the broader implications of the work for AI. Martha pushed me to communicate some of the research findings to the broader AI audience. It was her persistence and guidance that led to the JMLR publication.

My external examiner, Prof. Christopher Watkins, carefully read the thesis and shared insightful feedback during my doctoral exam. His positive response to the thesis was invaluable validation for me and has given me the confidence to further pursue the ideas introduced in this thesis.

I am grateful to my colleagues, family, and friends for making the PhD journey enjoyable. Some of them, in no particular order, are Arsalan, Yi, Abhishek,

Kris, Fernando, Kenny, Tian, Chen, Zaheer, Raksha, Suyug, Malvika, Banafshe, Sina, Kristen, Maliha, Anahita, Fatima, Adrian, Justin, Yuxin, Dawn, Aidan, Paritosh, Prabhat, David, Eric, Alex, Katy, Shibhansh, Anna, Scott, Adam, Martha S., Adrian, Calarina, Esraa, Farzane, Esra'a, Abdul, Han, Haseeb, Niko, Jiamin, Edan, Subhojeet, Erfan, Manan, Mohamed, Chunlok, Amir, Vincent, Jun, Vlad, Abdul, Kevin, Jacob, Hugo, Sam, Nicole, Sehrish, and Taha.

Finally, thank you Leticia for your love and support, and Rumi for forcing me to take breaks for walks.

Table of Contents

List of Tables

List of Figures

Chapter 1

Learning by Interacting with the World

Learning by interacting with the world is a powerful paradigm for building general-purpose autonomous systems. An agent can sense its environment through sensors, such as cameras and microphones, and take actions to influence its environment. It receives feedback from the environment—information about the influences of its actions—and can use this feedback to adapt its behavior. In the simplest case, the feedback can tell the agent how good the outcome of an action was. It is natural for the agent to adapt to repeat actions that led to good outcomes and avoid those that led to poor outcomes. More often the feedback is more nuanced: perhaps it tells the agent that the outcome of a sequence of actions was better than what the agent anticipated. Regardless of how clear the feedback is, as long as certain sequences of sensory inputs and actions are consistently correlated with good or bad outcomes, it can be used to adapt the future behavior of the agent. In other words, it can be used to learn.

It is easy to introspect and realize that we, humans, learn by interacting with the world. We are worse at a new video game or sport the first time we try it and get better over time, often without any explicit coaching. Navigating to an address for the first time is more effortful than navigating to the same address the second time. Navigating to an address we frequent requires almost no mental effort.

It is also evident that we learn immediately and continually. If we meet

someone new and see them again in an hour, we remember their face; missing a stop sign obstructed by trees once is sufficient to make us wary of the obstruction the next time we drive the same route. Learning is ingrained in our everyday lives, and we cannot switch off our ability to learn at will. An inability to learn in a person is considered a disability due to its debilitating effects on their everyday life.

Saying learning is continual is not the same as saying learning is permanent. A bounded learner can only learn so much before it has to discard some information. We, humans, forget information that is not reinforced or rehearsed. We don't remember our old addresses, phone numbers, email addresses, and license plate numbers. There is no simple rule that dictates when and what we forget. We don't remember many aspects of our lives for more than a day, and we remember many aspects for years. For example, most of us would be hard-pressed if asked to remember what we had for lunch two days ago and hardly any of us would remember our lunch from a week ago. At the same time, we would have no problem recalling the last concert or conference we attended, even if we attended it months ago.

The exact mechanisms of learning and forgetting in humans are unclear, but there is little doubt that we learn continually, and we forget gracefully. The holy grail of artificial intelligence researchers, in my view, should be algorithms that can enable agents to learn continually and forget gracefully.

Is it always essential to learn immediately and continually? In other words, are there problems for which agents can discover the optimal behavior once and remain unchanged for the rest of their lives? In certain problems, learning immediately and continually is not essential. A system that can play Chess or Go at a superhuman level does not need to continue to adapt to play against humans. The rules of these games are fixed and adaptation is not needed once an unbeatable policy has been discovered. In other problems learning continually and immediately is a necessity. These problems exist in large, ever-changing, and partially observable environments. I call these environments *big worlds* and hypothesize that many real-world problems involve achieving goals in big worlds—*the big world hypothesis*.

An agent living in a big world encounters new situations throughout its lifetime. To such an agent the world appears non-stationary and even previously seen situations can require adaptation. Any amount of experience is insufficient for all future predictions and continual learning is necessary for strong performance. Learning immediately is also advantageous in big worlds to minimize repeating poor behavior.

How close are we to building systems that can learn immediately and continually? We have made significant progress towards building algorithms that can learn complex behaviors from interaction by combining principles from the fields of deep learning and reinforcement learning. The resulting algorithms are collectively referred to as *deep RL*. Deep RL, while quite capable of learning sophisticated behavior for complex tasks, is ineffective for learning continually and immediately. It uses deep neural networks for learning which do not forget gracefully in supervised learning settings (French, 1999) and reinforcement learning settings (Kirkpatrick et al., 2017). Deep neural networks also lose the ability to learn over time (Dohare et al., 2024). Moreover, they rely on large amounts of computational resources only available in a special training phase and missing for the majority of the lifetime of the agents (*e.g.*, see the difference in resources used for training vs inference in works by Vinyals et al., 2019 and Berner et al., 2019).

In this dissertation, I propose several algorithms for learning in big worlds. My algorithms, developed in collaboration with others, can learn quickly and continually while only using resources available to the agent throughout its lifetime. They can be grouped into two categories: 1) algorithms for quick and robust linear learning, and 2) algorithms for learning non-linear recurrent features.

For quick and robust linear learning, I augment TD learning with three ideas that are 1) step-size optimization, 2) a bound to prevent updates that are too large, and 3) a mechanism to reduce step-size parameters when they are too large. I combine the three ideas into a single algorithm called SwiftTD. I then combine the same three ideas with Sarsa to get Swift-Sarsa.

For learning non-linear recurrent features I propose two ideas. First, I

show that by initializing recurrent features using experience, as opposed to initializing them with random weights, we can find useful recurrent features. Second, I show that by constraining the architecture of a recurrent network we can achieve unbiased and efficient gradient-based recurrent learning.

Chapter 2

Background

The solution methods introduced in this dissertation build upon earlier work on temporal-difference (TD) learning, step-size optimization, *real-time recurrent learning* (RTRL), and generate-and-test algorithms.

2.1 Notation

I use bold lowercase letters for real-valued vectors, for example, $\boldsymbol{x} \in \mathbb{R}^n$ is a vector with n components, and I use square brackets to refer to a component of a vector; for example, $x[i] \in \mathbb{R}$ is the ith component of the vector $\boldsymbol{x}$. I use subscripts to show time steps for time-dependent vectors and scalars, for example, $\boldsymbol{x}_t$, $x_t[i]$, and α_t. In some places, I use a list in the subscript for elements that are a function of variables from different time steps, for example, $y_{t_1,t_2} = x_{t_1} + x_{t_2}$.

2.2 Temporal Difference Learning

TD learning (Sutton, 1988) is an online and scalable mechanism for learning predictive knowledge. It is 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), Actor-Critic (Konda & Tsitsiklis, 2000), *etc.*

The key idea of TD learning is to learn from *bootstrapped targets* that are a combination of partial feedback and the difference between the agent's

Algorithm 1: TD(0) with linear function approximation

Hyperparameters: α
Initializations: $\boldsymbol{w} \leftarrow \mathbf{0} \in \mathbb{R}^n, \boldsymbol{\phi_{old}} = \mathbf{0} \in \mathbb{R}^n$, $v^{old} = 0 \in \mathbb{R}$
while *alive* **do**
  Receive $\boldsymbol{\phi}, \gamma$ and r
  $v \leftarrow \sum_{i=1}^{n} w[i]\phi[i]$
  $\delta \leftarrow r + \gamma v - v^{old}$
  for $i \in \{0, 1, \cdots, n\}$ **do**
    $w[i] \leftarrow w[i] + \alpha\delta\phi^{old}[i]$
  $\boldsymbol{\phi^{old}} \leftarrow \boldsymbol{\phi}$
  $v^{old} \leftarrow \sum_{i=1}^{n} w[i]\phi[i]$

subjective values of different situations. Using bootstrapped targets allows TD algorithms to learn online and incrementally without storing experience. One of the simplest algorithms for temporal difference learning is TD(0) with linear function approximation.

TD(0) with linear function approximation learns to predict the discounted sum of future values of a cumulant, r, by linearly combining a feature vector, $\boldsymbol{\phi}$, with a learnable weight parameter vector, $\boldsymbol{w}$. The sum of future values of r are discounted by the discount factor, γ.

Let $\boldsymbol{w}_{t-1} \in \mathbb{R}^n$, $\boldsymbol{\phi}_t \in \mathbb{R}^n$, γ and r_t be the weight parameter vector, the feature vector, the discount factor, and the cumulant at the start of time step t, respectively. The prediction made by TD(0) at time step t is

$$v_{t-1,t} = \sum_{i=1}^{n} w_{t-1}[i]\phi_t[i]. \tag{2.1}$$

The bootstrapped target for learning is $r_t + \gamma v_{t-1,t}$, which depends on the weight parameter vector. TD(0) ignores the impact of changing the weight parameter vector on the target in its learning update—*the semi-gradient assumption*. It learns by updating the prediction for the feature vector from one step ago to match the bootstrapped target. The prediction associated with the feature vector from one time step ago is:

$$v_{t-1,t-1} = \sum_{i=1}^{n} w_{t-1}[i]\phi_{t-1}[i], \tag{2.2}$$

Algorithm 2: TD(λ)

Hyperparameters: α and λ
Initializations: $\boldsymbol{w} \leftarrow \mathbf{0} \in \mathbb{R}^n, \boldsymbol{z} \leftarrow \mathbf{0} \in \mathbb{R}^n$, and $v^{old} = 0$
while *alive* **do**
  Receive $\boldsymbol{\phi}, \gamma$, and r
  $v \leftarrow \sum_{i=1}^{n} w[i]\phi[i]$
  $\delta \leftarrow r + \gamma v - v^{old}$
  for $i \in \{0, 1, \cdots, n\}$ **do**
    $w[i] \leftarrow w[i] + \alpha\delta z[i]$
    $z[i] \leftarrow \gamma\lambda z[i] + \phi[i]$
  end
  $v^{old} \leftarrow \sum_{i=1}^{n} w[i]\phi[i]$
end

and the *TD error* is

$$\delta_t = r_t + \gamma v_{t-1,t} - v_{t-1,t-1}. \tag{2.3}$$

TD(0) learns by minimizing the squared error between the prediction and the target and updates the ith parameter as:

$$w_t[i] = w_{t-1}[i] + \alpha\delta_t\phi_{t-1}[i], \tag{2.4}$$

where α is the step-size parameter. The pseudocode of TD(0) with linear function approximation is Algorithm 1.

Bootstrapped targets that only use one step of feedback are not always ideal. For many real-world problems, it is better to use targets that incorporate feedback from multiple time steps. Two examples of bootstrapped targets that use multiple time steps of feedback are n-step returns and λ-returns (Sutton & Barto, 2018).

2.3 n-step Return, λ-return and TD(λ)

The n-step return is defined as:

$$G_{t:t+n} \doteq r_{t+1} + \gamma r_{t+2} + \cdots + \gamma^{n-1} r_{t+n} + \gamma^n v_{t+n-1,t+n}, \tag{2.5}$$

where $v_{t+n-1,t+n}$ is the agent's prediction at time $t+n$ using the weight parameter vector at the end of time $t+n-1$.

Bootstrapped targets can be constructed by combining multiple bootstrapped targets to form *compound returns*. The λ-return is a special form of a compound return that combines n-step returns for all n weighted by a geometric series. For $\lambda \in [0, 1)$ it is defined as:

$$G_t^\lambda \doteq (1-\lambda) \sum_{n=1}^{\infty} \lambda^{n-1} G_{t:t+n}. \tag{2.6}$$

λ-returns are unique because the prediction error with respect to the λ-return can be written as a sum of td errors as

$$G_t^\lambda - v_{t-1,t} = \sum_{i=t+1}^{\infty} (\gamma\lambda)^{i-t-1} \delta_i. \tag{2.7}$$

TD(λ) (Sutton, 1988)—Algorithm 2—exploits the identity in Equation 2.7 to achieve online and incremental learning from λ-returns using an *eligibility trace* vector with the same number of components as the weight parameter vector. It updates the ith component of the eligibility trace vector as

$$z_t[i] = \gamma\lambda z_{t-1}[i] + \phi_t[i], \tag{2.8}$$

which is used to update the ith component of the weight parameter vector as

$$w_t[i] = w_{t-1}[i] + \alpha\delta_t z_{t-1}[i]. \tag{2.9}$$

TD(λ) was used by Tesauro (1995) to develop TD-Gammon, a program that learned to play backgammon well using reinforcement learning. Learning from λ-returns also provably improves convergence under the right conditions (Tsitsiklis & Van Roy, 1996).

The weight updates performed by TD(λ), however, are not identical to an algorithm learning directly from λ-returns. The identity in Equation 2.7 only holds when the weight parameter vector does not change over time. TD(λ) is a good approximation to learning from λ-returns when the step-size parameter is small.

TD(λ) uses a constant step-size parameter. A simple change gives us *TD(λ) with time-dependent step-size parameter*. Instead of using the step-size parameter in the weight update, TD(λ) with a time-dependent step-size parameter

Algorithm 3: TD(λ) with time-dependent step-size parameter

Hyperparameters: λ
Initializations: $\boldsymbol{w} \leftarrow \mathbf{0} \in \mathbb{R}^n, \boldsymbol{z} \leftarrow \mathbf{0} \in \mathbb{R}^n$, and $v^{old} = 0$
while *alive* **do**
  Receive $\alpha, \boldsymbol{\phi}$, γ, and reward r
  $v \leftarrow \sum_{i=1}^{n} w[i]\phi[i]$
  $\delta \leftarrow r + \gamma v - v^{old}$
  for $i \in \{0, 1, \cdots, n\}$ **do**
    $w[i] \leftarrow w[i] + \delta z[i]$
    $z[i] \leftarrow \gamma\lambda z[i] + \alpha\phi[i]$
  $v^{old} \leftarrow \sum_{i=1}^{n} w[i]\phi[i]$

scales the update to the eligibility trace vector with the step-size parameter. Let α_t be the step-size parameter at time t. Then TD(λ) with time-dependent step-size parameter updates the ith component of the eligibility trace vector as:

$$z_t[i] = z_{t-1}[i] + \alpha_t \phi_t[i], \tag{2.10}$$

and updates the ith component of the weight parameter vector as

$$w_t[i] = w_{t-1}[i] + \delta_t z_{t-1}[i]. \tag{2.11}$$

The pseudocode for TD(λ) with time-dependent step-size parameter is in Algorithm 3.

Algorithm 2 and 3 iterate over all components of the feature vector and the eligibility trace vector at every time step. If the feature vector or the eligibility trace vector are sparse, resources can be saved by skipping some computation. Some operations on TD(λ) can be skipped for zero components of the feature vector and some can be skipped for zero components of the eligibility trace vector. I call the algorithm that takes advantage of sparse feature vectors and eligibility trace vectors *TD(λ) with sparse computation*.

Algorithm 4 implements TD(λ) with sparse computation. It replaces the single loop over all components of vectors with two loops, one over non-zero components of the eligibility trace vector and one over non-zero components of the feature vector.

Algorithm 4: TD(λ) with time-dependent step-size parameter and sparse computation

Hyperparameters: λ
Initializations: $\boldsymbol{w} \leftarrow \mathbf{0} \in \mathbb{R}^n, \boldsymbol{z} \leftarrow \mathbf{0} \in \mathbb{R}^n$, and $v^{old} = 0$
while *alive* **do**
  Receive $\alpha, \boldsymbol{\phi}, \gamma$, and r
  $v \leftarrow \sum_{i|\phi[i]\neq 0} w[i]\phi[i]$
  $\delta \leftarrow r + \gamma v - v^{old}$
  for $i \mid z[i] \neq 0$ **do**
    $w[i] \leftarrow w[i] + \delta z[i]$ `// Update weight`
    $z[i] \leftarrow \gamma\lambda z[i]$ `// Decay eligibility trace`
  for $i \mid \phi[i] \neq 0$ **do**
    $z[i] \leftarrow z[i] + \alpha\phi[i]$ `// Update eligibility of the weight`
  $v^{old} \leftarrow \sum_{i|\phi[i]\neq 0} w[i]\phi[i]$

2.4 True Online TD(λ)

TD(λ) approximates the algorithm that learns from λ-returns. When the step-size parameter is small, it is a good approximation. When it is large, the error in the approximation can be significant. Van Seijen et al. (2016) proposed True Online TD(λ) to address the approximation error of TD(λ). True Online TD(λ) performs the same updates as the Online λ-return algorithm (Sutton & Barto, 2018).

True Online TD(λ) updates the ith component of the eligibility trace vector as:

$$z_t[i] = \gamma\lambda z_{t-1}[i] + \alpha\phi_t[i](1 - b_t), \tag{2.12}$$

where:

$$b_t = \lambda\gamma \sum_{i|\phi[i]\neq 0} z[i]\phi[i]. \tag{2.13}$$

It does not use the standard TD error, δ, but a slightly different term,

$$\delta'_t = r_t + \gamma v_{t-1,t} - v_{t-2,t-1}, \tag{2.14}$$

in its updates. The difference between δ and δ' is in the time indices of the prediction term.

Algorithm 5: True Online TD(λ) with time-dependent step-size parameter by Van siejen et al. (2016)

Hyperparameters: α and λ
Initializations: $(\boldsymbol{w}, \boldsymbol{\phi^{old}}, \boldsymbol{z}) \leftarrow (\mathbf{0}, \mathbf{0}, \mathbf{0})$, and $v^{old} = 0$
while *alive* **do**
  Receive $\boldsymbol{\phi}$,γ, and r
  $v \leftarrow \sum_{i=1}^{n} w[i]\phi^{old}[i]$
  $v' \leftarrow \sum_{i=1}^{n} w[i]\phi[i]$
  $\delta' \leftarrow r + \gamma v - v^{old}$
  for $i \in \{1, 2, \cdots, n\}$ **do**
    $z[i] \leftarrow \gamma\lambda z[i] + \alpha\phi^{old}[i](1 - \gamma\lambda(\boldsymbol{z}^T\boldsymbol{\phi}))$
    $w[i] \leftarrow w[i] + \delta' z[i] - \alpha\phi^{old}[i](v - v^{old})$
  $\boldsymbol{\phi^{old}} \leftarrow \boldsymbol{\phi}$
  $v^{old} \leftarrow v$

It updates the ith components of the weight parameter vector as:

$$w_t[i] = w_{t-1}[i] + \delta'_t z_{t-1}[i] - \alpha\phi_{t-1}[i](v_{t-1,t-1} - v_{t-2,t-1}). \tag{2.15}$$

It can be extended to use time-dependent step-size parameters. Algorithm 5 is the pseudocode of True Online TD(λ) with time-dependent step-size parameters and is the same as by Van Seijen et al. (2016) (See Algorithm 4 in their paper). Similar to TD(λ), True Online TD(λ) can be implemented by only iterating over non-zero components of the feature vectors and eligibility trace vectors. Algorithm 6 is True Online TD(λ) with sparse computation.

Mapping Algorithm 6 to Algorithm 5 takes some effort. The scalar v^{δ} in Algorithm 6 is the same as the scalar $(v - v^{old})$ in Algorithm 5. The scalar b in Algorithm 6 is the same as $\gamma\lambda(\boldsymbol{z}^T\boldsymbol{\phi})$. The term $\gamma\lambda$ is missing when estimating b because, by the time b is estimated, the eligibility vector has already been multiplied by $\gamma\lambda$.

2.5 Step-size Optimization

The step-size parameter is an important hyperparameter of a learning algorithm. If it is too small, learning can be slow. If it is too large, learning can be unstable, and it can diverge. The optimal value of the step-size parameter is problem-dependent and can change over time.

Algorithm 6: True Online TD(λ) with time-dependent step-size parameter and sparse computation

Hyperparameters: α and λ
Initializations: $(\boldsymbol{w}, \boldsymbol{z^\delta}, \boldsymbol{z}) \leftarrow (\mathbf{0}, \mathbf{0}, \mathbf{0})$, and $(v^\delta, v^{old}) = (0, 0)$
while *alive* **do**
  Receive $\boldsymbol{\phi}$, γ, and r
  $v \leftarrow \sum_{i|\phi[i]\neq 0} w[i]\phi[i]$
  $\delta' \leftarrow r + \gamma v - v^{old}$
  for $i \mid z[i] \neq 0$ **do**
    $\delta^w[i] \leftarrow \delta' z[i] - z^\delta[i] v^\delta$
    $w[i] \leftarrow w[i] + \delta^w[i]$
    $z^\delta[i] = 0$
    $z[i] \leftarrow \gamma\lambda z[i]$
  $v^\delta \leftarrow 0$
  $b \leftarrow \sum_{i|\phi[i]\neq 0} z[i]\phi[i]$
  for $i \mid \phi[i] \neq 0$ **do**
    $v^\delta \leftarrow v^\delta + \delta^w[i]\phi[i]$
    $z^\delta[i] \leftarrow \alpha\phi[i]$
    $z[i] \leftarrow z[i] + z^\delta[i](1 - b)$
  $v^{old} \leftarrow v$

Using a scalar step-size parameter for updating all components of the weight parameter vector can be limiting. Some features are more important for learning than others, and it can be beneficial to update the weight parameters associated with them with larger step-size parameters. On the other hand, weight parameters associated with some features don't have to change once learned, and it can be beneficial to reduce the step-size parameters in their updates over time. The degree of noise can also vary across features requiring different step-size parameters when updating different weight parameters.

IDBD (Sutton, 1992) is a supervised learning algorithm that uses a step-size parameter vector to overcome the limitations of sharing a scalar step-size parameter, and it automatically finds a good value of the step-size parameter vector by meta-learning. It uses gradient-based meta-learning and incrementally approximates the gradients of the step-size parameters using forward-view differentiation (Williams & Zipser, 1989). Intuitively, IDBD increases step-size parameters associated with features that generalize well to future examples.

Let $\boldsymbol{w}$ be the weight parameter vector and $\boldsymbol{\beta}$ be the step-size parameter vector of IDBD. To update the ith component of the weight parameter vector, IDBD uses $e^{\beta[i]}$ instead of α. If $\boldsymbol{\phi}_t \in \mathbb{R}^n$ is the feature vector at time t and y^*_{t+1} is the target associated with this feature vector, then the prediction made by IDBD is

$$y_t = \sum_{i=1}^{n} \phi_t[i] w_t[i], \tag{2.16}$$

and the ith component of the step-size parameter vector is updated as

$$\beta_{t+1}[i] = \beta_{t+1}[i] + \theta \left(y^*_{t+1} - y_t\right) \phi_t[i] h_t[i], \tag{2.17}$$

where θ is the *meta-step-size parameter*. $h[i]$ is initialized to be zero and $h_t[i]$ is estimated as

$$h_t[i] = h_{t-1}[i] \left(1 - e^{\beta_t[i]} \phi_{t-1}[i]^2\right) + e^{\beta_t[i]} \left(y^*_t - y_{t-1}\right) \phi_{t-1}[i]. \tag{2.18}$$

Sutton (1992) showed that $h_t[i]$ approximates the meta-gradient $\frac{\partial w_t[i]}{\beta[i]}$ under the assumption that $\beta[i]$ is not updated during learning. However, IDBD updates $\beta[i]$ using Equation 2.17. The updates to $\beta[i]$ introduce further approximation error to an already approximate estimate of the meta-gradient.

IDBD updates the ith component of the weight parameter vector as

$$w_{t+1}[i] = w_t[i] + e^{\beta_{t+1}[i]} \left(y^*_{t+1} - y_t\right) \phi_t[i]. \tag{2.19}$$

IDBD is fundamentally different from popular adaptive step-size algorithms such as RMSProp (Tieleman & Hinton, 2012) and Adam (Kingma & Ba, 2015). Degris, Javed, Sharifnassab, Liu, & Sutton (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 for which they analytically computed the step-size parameter vector that achieved the best performance. They showed that updates done by IDBD moved the step-size parameter vector towards the optimal step-size parameter vector whereas step-size normalization, as done by RMSProp, did not.

2.6 Feature Generation in Deep Learning

Using observational data linearly is rarely sufficient for learning complex predictions and behaviors. To do well in general problems it can be necessary to have features that are complex functions of the history of the agent's observations.

The dominant paradigm for learning complex features is *deep learning* (Lecun, Bengio, & Hinton, 2015). Deep learning is a collection of ideas for training artificial neural networks with many layers to perform well on supervised learning problems. In deep learning, features are generated in three distinct stages.

In the first stage, the functional form of the features is designed by human experts. The design incorporates prior knowledge, for example, translation invariance of convolutional neural networks (Lecun et al., 1988); ease of optimization, for example, skip connections (He et al., 2016) and batch-normalization (Santurkar et al., 2018); and hardware constraints, for example, transformers designed to be parallelizable (Vaswani et al., 2017). All functional forms have some parameters that can be learned.

In the second stage, the learnable parameters are initialized by sampling them from some distribution. Generally, the parameters are initialized to small values, and special care is taken to ensure that the scales of gradients of the parameters are similar in different layers.

In the third and final step, the learnable parameters are updated using gradient descent. The gradients are computed using back-propagation (Rumelhart et al., 1986) for feedforward networks and using back-propagation through time (BPTT) (Werbos, 1988 and Robinson & Fallside, 1987) for recurrent networks. Some form of gradient normalization, such as RMSProp (Tieleman & Hinton, 2012) or Adam (Kingma & Ba, 2015), is used to make the scale of the updates to parameters in different layers comparable.

2.7 Feature Removal in Deep Learning

Feature removal is the idea of measuring the importance of features for the predictions and behaviors learned by the agent and removing those that are not useful. Most existing works do not look at feature removal as a continual process. The common paradigm is to train a large model and remove the useless features and parameters after training has finished. An important question for feature removal is to decide how to measure the usefulness of different features.

Neural network researchers have proposed multiple algorithms for estimating the importance of features or parameters for predictions and behaviors (for a detailed overview see Blalock et al., 2020). For example, Han et al. (2015) proposed pruning the parameters with the smallest magnitudes, and LeCun (1989) proposed approximating the impact of different parameters on predictions and pruning the features with the smallest impact. The general idea of these methods is to rank the parameters or features of a deep network and discard the least useful ones to get a smaller network that can be deployed more efficiently.

A handful of papers have explored the idea of online and continual pruning to improve performance. One family of works is called *dynamic sparse training* (DST) (see work by Mocanu et al., 2018 and Evci et al., 2020). In DST, features are pruned based on the magnitude of their outgoing weights and replaced with new features with randomly initialized weights. An independently evolving body of work under the umbrella term Generate & Test (G&T) algorithms has also looked at continual feature replacement as a mechanism for learning (Mahmood, 2017; Mahmood & Sutton, 2013). Both DST and G&T algorithms improve performance on supervised learning tasks. More recent work has discovered that deep neural networks lose their ability to learn over time (Dohare et al., 2024). They call this phenomenon the *loss of plasticity* and showed that continually replacing useless features with new random features mitigates the loss of plasticity.

Chapter 3

Problem Formulations

The goal of an agent learning from an online stream of data for predicting or controlling the future can be formalized as the lifetime performance of the agent on the prediction or the control problem.

3.1 The Prediction Problem

The prediction problem consists of observations and predictions. The agent receives an observation vector $\boldsymbol{x}_t \in \mathbb{R}^n$ and a discount factor γ_t at time step t and makes a scalar prediction $v_t \in \mathbb{R}$. The target for evaluating the prediction is computed by summing the future values of a scalar called the *cumulant* discounted by the discount factors. The cumulant can be any component of the observation vector with a fixed index. 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 $\boldsymbol{x}_t$, be the cumulant at time step t. The lifetime error is defined as:

$$\text{Lifetime error}(T) = \frac{1}{T}\sum_{t=1}^{T}\left(v_t - \sum_{j=t+1}^{T}\gamma_j^{j-t-1}r_j\right)^2, \tag{3.1}$$

where T is the lifetime parameter of the agent and is part of the problem. 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 a 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.

3.2 The Control Problem

The control problem consists of observations and actions. The agent perceives an observation vector $\mathbf{x}_t \in \mathbb{R}^n$ at time step t. It outputs an action vector $\boldsymbol{a}_t \in \mathbb{R}^d$. A special component of the observation vector is the reward, r_t. The index of the component that is the reward is fixed throughout the lifetime of the agent. Performance on a control problem is measured using the lifetime average reward defined as

$$\text{Lifetime reward}(T) = \frac{1}{T}\sum_{t=1}^{T} r_t. \tag{3.2}$$

In a control problem, the actions chosen by the agent control what observations the agent perceives in the future, and the agent seeks to maximize its lifetime reward by controlling its future.

Chapter 4

The Big World Hypothesis and its Ramifications

The *big world hypothesis* says that in many decision-making problems the agent is orders of magnitude smaller than the environment. It can neither fully perceive the state of the world nor can it represent the value or optimal action for every state. Instead, it must learn to make sound decisions using its limited understanding of the environment. The key research challenge for achieving goals in big worlds is to come up with solution methods that efficiently use the limited resources of the agent.

An opposing view to the big world hypothesis is that real-world decision-making problems have simple solutions. The agent is not only capable of representing the simple solution but also has additional capacity that can be used to search for the solution more efficiently—it is *over-parameterized*. The key research challenge for achieving goals with over-parameterized agents is to find solutions that enable optimal decision-making in perpetuity.

There are many problems that satisfy the big world hypothesis and many that do not. The problem of finding roots of a second-degree polynomial admits a simple solution that always works. Representing the value function of the game of Go for all states does not have a simple solution. The big world hypothesis is more a statement about the class of problems we should care about than a fact about all decision-making problems. It can be made true or false by exercising control over the design of the environment and the agent (*e.g.*, when developing benchmarks).

Developing algorithms for big worlds poses unique challenges. The best algorithms for big worlds might prefer fast approximate solutions over slow exact ones. They might learn incorrect simplistic models that are sufficient for achieving the agent's goals over causally correct complex models (*e.g.*, Newtonian physics as opposed to quantum mechanics). They might forgo knowledge that is not frequently used by the agent to make room for knowledge used more often. Such trade-offs do not exist for over-parameterized agents.

The big world hypothesis is not a novel proposition. Over the past few years, several independent works have entertained the idea of small-bounded agents learning in large unbounded environments. Sutton (2020) argued that the world is large and complex and an agent cannot learn everything there is to learn exactly. He proposed embracing function approximation for learning values, policies, models, and states. Dong et al. (2022) theoretically studied the performance of a reinforcement learning algorithm without making simplifying assumptions about the environment. Their work shifts the focus from making assumptions about the environment to making assumptions about the capabilities of the agent. Javed et al. (2023) empirically studied the performance of small agents in large environments. They found that approximate algorithms that use less computation can outperform exact algorithms that use more computation in big worlds. Kumar et al. (2023) showed that continual learning is a necessary element of reinforcement learning when the agent is computationally constrained.

Is the big world hypothesis a temporary artifact of the limitations of our current computers? Or would it have relevance even as computational resources grow? In the next section, I argue that the big world hypothesis is here to stay irrespective of the rate at which computational resources grow.

4.1 The Big World Hypothesis and Exponentially Growing Computation

Historically access to computation has increased exponentially. With continuing growth computers of the future could be sufficiently powerful to solve all

problems we care about using over-parameterized agents. I see two problems with this view.

First, it is not just our agents that are constrained by compute. The sensors used by our agents are also constrained by compute. A rise in computation makes it possible to sense the world with more precision and at a higher frequency. For example, within the last decade the camera sensors in our phones have gone from sensing 640 x 420 pixels at 30 fps—around 7 million pixels per second—to sensing in 4k at 60 fps—around 500 million pixels per second. To put these numbers in perspective, a modern smartphone camera sensor in 2024 can generate more data in a week than that used to train GPT-3 (Brown et al., 2020). Even with these massive increases in the ability to sense the world, our agents are not even close to sensing the world at its full scale. I speculate that as computational resources grow so would the appetite to sense the world at higher fidelity, making the decision-making problem more challenging.

The second problem with waiting for compute to grow is that as compute becomes more readily available, the world itself becomes more complex. From the perspective of an agent, the world consists of everything outside of itself. This includes other equally complex agents and computers. An agent that interacts with multiple other agents of similar capabilities would be unable to model the world exactly regardless of the rate at which computation grows.

A concrete example of the world getting more complex as computation grows is that of an agent playing the game of Go against an opponent. If the opponent picks moves randomly, then it is fairly simple for the agent to model the environment exactly. The dynamics of the environment can be simulated with a short program. However, if the opponent is more complex, such as an AlphaZero (Silver et al., 2015) agent, then the only way to model the dynamics of the environment correctly is to be able to represent the policy of the large AlphaZero agent accurately.

As computational resources increase so does the complexity of the world. The big world hypothesis is not a temporary artifact of the limitations of our current computers. For many problems, the world will always be much larger than any single agent.

4.2 Evidence Consistent with the Big World Hypothesis

There is some indirect evidence that the behavior of our learning algorithms on large problems is consistent with the big world hypothesis.

Silver et al. (2017) trained a large neural network to learn the value function for the game of Go. They found that even after extensive training the performance of the system could be improved if the decisions were taken by combining the value function with a planner.

If the neural network had the capacity to represent the optimal value function of Go, and it had been trained for a sufficiently long time, then decision-time planning should not have improved performance. Perhaps the neural network did not have sufficient capacity to represent the value function correctly for all states and the planner was able to fill in the gaps.

The second and more direct evidence comes from the work of Brown et al. (2020). They showed a clear trend between the model size and performance of neural networks when fitting large language datasets. They found that the train error and validation error on the dataset could be reduced by increasing the number of parameters in the network. Their findings make little sense if the neural networks were over-parameterized.

Neither of the two papers directly set out to test the big world hypothesis and their results have other explanations. However, they don't contradict it and provide circumstantial evidence for its relevance.

4.3 Ramifications of the Big World Hypothesis

The big world hypothesis is only worth discussing if accepting it would directly impact how we do research in AI. In the next subsections, I discuss three ways accepting the hypothesis can influence research today.

Online continual learning for achieving goals in big worlds

The need for online continual learning in big worlds is intuitive—if the agent does not have the resources to learn and retain everything important about the world simultaneously, then it can learn aspects that are important for decision-making at the current time and discard them when they are no longer. In the over-parameterized setting, on the other hand, there is no need for online continual learning. Once the agent has found the underlying optimal solution it can use it forever without changing.

Learning things when they are needed and discarding them when they are not is sometimes called *tracking*. Tracking has been empirically demonstrated to be superior to fixed solutions in partially observable environments by Sutton, Koop, & Silver (2007) and Silver, Sutton, & Müller (2008).

A key requirement for tracking to be effective is *temporal coherence*. Temporal coherence means that parts of the world the agent experiences from one step to the next are correlated. An agent learning online can exploit the temporal coherence to direct its resources to learn about the states of the world that are temporally close at the expense of those that are far away. Tracking can be a powerful solution method in temporally coherent big worlds.

Humans extensively rely on tracking in everyday life
Humans are continually learning agents. We extensively rely on tracking to achieve our goals. An intuitive example is that of exams. Given the choice between taking exams of different subjects on different days or taking them all on the same day, most of us would pick the former. Intuitively, it feels easier to have to only have to learn and remember the material for one exam at a time. This is exactly the behavior we should expect from a tracking agent in a big world.

An analogy of a tracking system is the cache used by a CPU. The cache is much smaller than the memory and can only store a small fraction of instructions and data used by a program. However, by retaining the right pieces of information and discarding the least useful ones, a small cache can have a high hit ratio. A high hit ratio is only possible when a program accesses memory predictably, akin to having temporal coherence in big worlds.

If we are to accept the hypothesis then we have to develop algorithms that can learn online and continually. This is a significant departure from the current practice of training agents offline and then deploying them.

Need for Computationally Efficient Learning Algorithms

In big worlds, increasing the size of the agent can improve performance. This raises an important trade-off between the complexity of the learning algorithm and the size of the agent. A trivial example is the mini-batch size of a deep RL algorithm, such as DQN (Mnih et al. 2015). For a fixed amount of resources, an agent can double the number of parameters by halving the mini-batch size.

Javed, Shah, Sutton, & White (2023) empirically demonstrated that approximate but efficient learning algorithms can outperform computationally expensive exact algorithms in big worlds. In their experiments, they evaluated tiny recurrent networks on the Arcade Learning Environment (Bellemare et al., 2013). They constrained all algorithms to use the same amount of per-step computation and found that a simple algorithm that used less computation was able to outperform a more complex algorithm by repurposing the saved computation to increase the size of the network.

> Accepting the big world hypothesis means we should actively look for more efficient learning algorithms.

Benchmarking in Big Worlds

A common way to evaluate algorithms is to run them on a standardized benchmark. A good benchmark is an accurate proxy for the problem we care about and allows us to do careful experiments. Designing a benchmark

for big worlds requires a different approach than designing a benchmark for over-parameterized agents.

One way to evaluate algorithms for big worlds is to test them on complex environments so that even our largest agents on the latest hardware are not over-parameterized. While this approach has merit, it makes it difficult to do careful and reproducible experiments.

The alternative is to restrict the computational capabilities of the agents instead of making the environments larger. The primary limitation of restricting agents is that we might miss out on emergent properties of large agents. However, a small agent learning in a non-trivial environment is still a better proxy for learning in big worlds than a large over-parameterized agent learning in the same environment.

Example: A typical DQN agent for Atari users orders of magnitude more computation than the environment.
Arcade learning environment (Bellemare et al., 2013) is a popular benchmark for reinforcement learning. A typical game in the benchmark can run at around 7000 frames per second on a modern CPU core. A DQN agent (Mnih et al., 2014), on the other hand, runs at 300 frames per second on a modern GPU. While it is hard to directly compare different implementations of the agent and the environment running on different hardware, it is clear that the agent uses orders of magnitude more computation than the environment in this case.

Restricting the computational capabilities of the agents is not trivial. There is no consensus on what aspects of the agents should be restricted. We could restrict the number of operations, the amount of memory, the amount of memory bandwidth, or the amount of energy the agent can use. The choice of constraints can have a significant impact on algorithms that win.

One option is to match the constraints on the agent with the constraints imposed by current computers. For example, if memory is cheaper than CPU cycles, then we might want to restrict the CPU cycles. Alternatively, if accessing the memory is a bottleneck, then we might want to restrict the memory bandwidth.

A second option is to limit energy usage. Energy is a universal constraint

that can take into account the evolution of hardware over time and can even drive research for designing better hardware for our agents. The downside of using energy as a constraint is that it is difficult to measure. Normally the computer running the agent is also running the environment, an operating system, and other unrelated processes, and isolating the energy used by the agent from background tasks is challenging.

The big world hypothesis has direct implications on what we choose to study and how we evaluate our algorithms. It is not a temporary artifact of the current limitations of our computers. It is imperative that we develop algorithms that can allow agents to achieve goals in big worlds. This requires developing computationally efficient algorithms for learning continually and rethinking the way we benchmark our algorithms.

Part I

Fast and Robust Linear Learning

Chapter 5

Temporal Difference Learning with Step-size Optimization

Existing algorithms for TD learning can be ineffective for learning incrementally and quickly. They force us to make one of the following three unsatisfactory choices. First, we could use them to 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 them using a large step-size parameter. Doing so could result in faster learning but risks divergence. Third, we could use them to 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, using every sample in multiple learning updates is computationally wasteful, and it makes agents less reactive—feedback is not reflected in their predictions and behaviors immediately.

An alternative and computationally efficient solution is to use a combination of large and small step-size parameters. Each component of the weight parameter vector can be updated using its own step-size parameter. If the agent can set the step-size parameters to small values for features that are not correlated with the prediction error and to large values for features that are correlated with the prediction error, it could learn quickly without risking divergence. The challenge is to find the right step-size parameters for different features.

A promising solution for setting different step-size parameters for different features is to learn them. IDBD (Sutton, 1992) does that for linear regression. We propose algorithms for learning step-size parameters for TD learning.

Three prior works have extended 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). Young et al. (2019), independently of my work, correctly extended IDBD to TD(λ). The extension of IDBD to TD(λ) in this thesis is identical to that by Young et al. (2019); the extension to True Online TD(λ) is novel.

In the following sections I derive the update rules for computing the meta-gradient of the step-size parameters for TD(λ) and compare the resulting algorithm—*TD(λ) with step-size optimization*—with TD(λ) and TIDBD(λ) (Kearney et al., 2018). I then derive the update rules for computing the meta-gradient for True Online TD(λ) and compare the resulting algorithm—*True Online TD(λ) with step-size optimization*—with True Online TD(λ). Finally, I compare TD(λ) with step-size optimization and True Online TD(λ) with step-size optimization.

5.1 TD(λ) with Step-size Optimization

TD(λ) with step-size optimization uses the λ-return as the target defined as:

$$G_t^\lambda = (1-\lambda)\sum_{n=1}^{\infty}\lambda^{n-1}G_{t:t+1}. \tag{5.1}$$

IDBD parameterizes the step-size parameters with a vector $\boldsymbol{\beta}$, and it updates the ith weight parameter with the step-size parameter $e^{\beta[i]}$. TD(λ) with step-size optimization uses the same parameterization for the step-size parameters. It predicts the value at time step t as:

$$v_{t-1,t} = \sum_{i=1}^{n} w_{t-1}[i]\phi_t[i], \tag{5.2}$$

and updates the ith weight parameter as:

$$w_t[i] = w_{t-1}[i] + \delta_t z_{t-1}[i], \tag{5.3}$$

where:

$$\delta_t = r_t + \gamma_t v_{t-1,t} - v_{t-1,t-1}, \tag{5.4}$$

and $z_t[i]$ is updated as:

$$z_t[i] = \gamma_t \lambda z_{t-1}[i] + e^{\beta_t[i]} \phi_t[i]. \tag{5.5}$$

The step-size parameters are updated to minimize the squared error between the prediction and the λ-return. The meta-gradient of the squared error with respect to the ith step-size parameter is:

$$\begin{aligned}
\frac{1}{2}\frac{\partial \mathcal{L}(t)}{\partial \beta[i]} = \frac{\partial}{\partial \beta[i]} \frac{(G_t^\lambda - v_{t-1,t})^2}{2} &= -(G_t^\lambda - v_{t-1,t}) \frac{\partial v_{t-1,t}}{\partial \beta[i]} \\
&= -(G_t^\lambda - v_{t-1,t}) \frac{\partial}{\partial \beta[i]} \sum_{j=1}^{n} w_{t-1}[j] \phi_t[j] \\
&= -(G_t^\lambda - v_{t-1,t}) \sum_{j=1}^{n} \phi_t[j] \frac{\partial w_{t-1}[j]}{\partial \beta[i]}.
\end{aligned} \tag{5.6}$$

Similar to Sutton (1992), I assume that the indirect impact of $\beta[i]$ on $w[j]$ for $j \neq i$ is negligible, as changing $e^{\beta[i]}$ will mostly impact the weight parameter $w[i]$. For a more detailed discussion on this approximation, see work by Javed et al. (2021). The approximation simplifies Equation 5.6 as:

$$(G_t^\lambda - v_{t-1,t}) \sum_{j=1}^{n} \phi_t[j] \frac{\partial w_{t-1}[j]}{\partial \beta[i]} \approx (G_t^\lambda - v_{t-1,t}) \phi_t[i] \frac{\partial w_{t-1}[i]}{\partial \beta[i]}. \tag{5.7}$$

I define $h_{t-1}[i]$ to be $\frac{\partial w_{t-1}[i]}{\partial \beta[i]}$. Then:

$$\begin{aligned}
h_{t-1}[i] &= \frac{\partial w_{t-1}[i]}{\partial \beta[i]} \\
&= \frac{\partial \left(w_{t-2}[i] + \delta_{t-1} z_{t-2}[i] \right)}{\partial \beta[i]} \\
&= \frac{\partial w_{t-2}[i]}{\partial \beta[i]} + \frac{\partial (\delta_{t-1} z_{t-2}[i])}{\partial \beta[i]} \\
&= h_{t-2}[i] + z_{t-2}[i] \frac{\partial \delta_{t-1}}{\partial \beta[i]} + \delta_{t-1} \frac{\partial z_{t-2}[i]}{\partial \beta[i]}.
\end{aligned} \tag{5.8}$$

The gradient $\frac{\partial \delta_{t-1}}{\partial \beta[i]}$ is:

$$\begin{aligned}\frac{\partial \delta_{t-1}}{\partial \beta[i]} &= \frac{\partial(-\sum_{j=1}^{n} w_{t-2}[j]\phi_{t-2}[j])}{\partial \beta[i]} \\ &\approx -h_{t-2}[i]\phi_{t-2}[i]\end{aligned} \tag{5.9}$$

Finally, I define $\bar{z}_{t-2}[i]$ as $\frac{\partial z_{t-2}[i]}{\partial \beta[i]}$. Then:

$$\begin{aligned}\bar{z}_{t-2}[i] &= \frac{\partial}{\partial \beta[i]}\left(\gamma_{t-2}\lambda z_{t-3}[i] + e^{\beta[i]}\phi_{t-2}[i]\right) \\ &= \gamma_{t-2}\lambda \bar{z}_{t-3}[i] + e^{\beta[i]}\phi_{t-2}[i] \\ &= z_{t-2}[i].\end{aligned} \tag{5.10}$$

The final recursive update rule for $h_{t-1}[i]$ is:

$$\begin{aligned}h_{t-1}[i] &\approx h_{t-2}[i] + z_{t-2}[i]h_{t-2}[i]\left(\gamma_{t-1}\phi_{t-1}[i] - \phi_{t-2}[i]\right) + \delta_{t-1}z_{t-2}[i] \\ &= h_{t-2}[i]\left(1 - z_{t-2}[i]\phi_{t-2}[i]\right) + \delta_{t-1}z_{t-2}[i].\end{aligned} \tag{5.11}$$

We still need to estimate the error term, $G_t^\lambda - v_{t-1,t}$, in Equation 5.6 incrementally. It can be approximated as the sum of TD errors as:

$$G_t^\lambda - v_{t-1,t} \approx \sum_{i=t+1}^{\infty} (\gamma_i\lambda)^{i-t-1}\delta_i, \tag{5.12}$$

where equality holds if the weight parameters are kept fixed over time. The final meta-gradient of the squared error w.r.t $\beta[i]$ is:

$$\begin{aligned}\frac{\partial \mathcal{L}(t)}{\beta[i]} &\approx \left(\sum_{j=t+1}^{\infty} (\gamma_j\lambda)^{j-t-1}\delta_j\right) h_{t-1}[i]\phi_t[i] \\ &= \left(\sum_{j=t+1}^{\infty} (\gamma_j\lambda)^{j-t-1} h_{t-1}[i]\phi_t[i]\delta_j\right) \\ &= h_{t-1}[i]\phi_t[i]\delta_{t+1} + \gamma_j\lambda\ h_{t-1}[i]\phi_t[i]\delta_{t+2} + \gamma_j^2\lambda^2 h_{t-1}[i]\phi_t[i]\delta_{t+3} + \ldots,\end{aligned} \tag{5.13}$$

and it is used to update $\beta[i]$ at time step t as:

$$\beta_t[i] = \beta_{t-1}[i] + \frac{\theta}{e^{\beta_{t-1}[i]}}\delta_t p_{t-1}[i], \tag{5.14}$$

where $p[i]$ is initialized to 0 and $p_t[i]$ updated as:

$$p_t[i] = \lambda\gamma_t p_{t-1}[i] + \phi_t[i]h_{t-1}[i]. \tag{5.15}$$

The meta-update in Equation 5.14 uses $\frac{\theta}{e^{\beta_{t-1}[i]}}$ to scale the meta-gradient, where θ is a hyperparameter called the *meta-step-size*, and scaling by $\frac{1}{e^{\beta_{t-1}[i]}}$ makes the scale of the meta-gradient invariant to the magnitude of the step-size parameter.

TD(λ) with step-size optimization has two important hyperparameters, the value of the step-size parameters at initialization (α^{init}) and the meta-step-size parameter (θ). The pseudocode for TD(λ) with step-size optimization is Algorithm 7.

Algorithm 7: TD(λ) with Step-size Optimization

Hyperparameters: $\alpha^{init}, \lambda, \theta$
Initializations: $(\boldsymbol{w}, \boldsymbol{z}) \leftarrow (\mathbf{0}, \mathbf{0}) \in \mathbb{R}^n, \boldsymbol{\beta} = \boldsymbol{ln}(\boldsymbol{\alpha^{init}}) \in \mathbb{R}^n$
while *alive* **do**
  Receive $\boldsymbol{\phi}$, γ, and r
  $\delta \leftarrow r + \gamma \sum_{i|\phi[i]\neq 0} w[i]\phi[i] - \sum_{i|\phi^{old}[i]\neq 0} w[i]\phi^{old}[i]$
  for $i \mid z_i \neq 0$ **do**
    $w[i] \leftarrow w[i] + \delta z[i]$
    $\beta[i] \leftarrow \beta[i] + \frac{\theta}{e^{\beta[i]}}\delta p[i]$
    $h^{old}[i] \leftarrow h[i]$
    $h[i] \leftarrow h^{temp}[i] + z[i]\delta$
    $(z[i], p[i]) \leftarrow (\gamma\lambda z[i], \gamma\lambda p[i])$
  for $i \mid \phi[i] \neq 0$ **do**
    $z[i] \leftarrow z[i] + e^{\beta[i]}\phi[i]$
    $p[i] \leftarrow p[i] + \phi[i]h^{old}[i]$
    $h^{temp}[i] \leftarrow h[i](1 - z[i]\phi[i])$
  $\boldsymbol{\phi^{old}} \leftarrow \boldsymbol{\phi}$

5.2 Comparing TD(λ) with TIDBD(λ)

Kearney et al. (2018) used a different meta-gradient for learning the step-size parameters of TD(λ). Their meta-gradient is:

$$\frac{\partial}{\beta[i]} \frac{(G_{t:t+1} - v_{t-1,t})^2}{2}, \tag{5.16}$$

where $G_{t:t+1}$ is:

$$G_{t:t+1} = r_{t+1} + \gamma_{t+1} v_{t,t+1}. \tag{5.17}$$

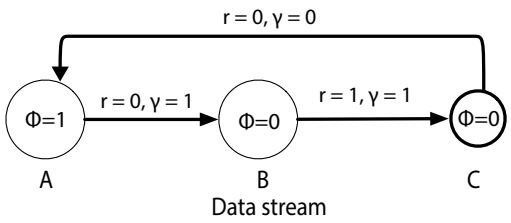

Figure 5.1: The data stream consists of a single feature that is 1, 0, 0, 1, 0, $\cdots$. γ is zero when going from C to A and one otherwise. An agent learning with a simple weight parameter w using TD(λ) should converge to $w = 1$ as the value of state A is 1 and w has no influence on the predictions in states B and C. An agent using TD(0), on the other hand, should converge to $w = 0$.

Using $G_{t:t+1}$ as the target in the meta-gradient is an odd choice because TD(λ) uses λ-returns as targets and not the one-step returns. I elucidate the problem with a simple experiment.

The experiment uses an environment with three states—A, B, and C. The agent starts in A and deterministically transitions from A to B, B to C, and C to A. The cycle continues indefinitely. The value of γ is 0 when transitioning from C to A and one everywhere else. The reward is 1 when transitioning from B to C and 0 everywhere else. Each state has a scalar feature. State A, B, and C have features 1, 0, and 0, respectively. The environment is shown in Figure 5.1.

The agent has a scalar weight parameter initialized to zero and a scalar step-size parameter, β, initialized to -10; λ is one.

The problem is constructed such that the weight parameter, w, would converge to 0 if it is updated using TD(0) and to 1 if it is updated using TD(λ) with $\lambda = 1$.

Experiment and Results

I used TD(λ) with step-size optimization and TIDBD(λ) to learn for five million steps on the above mentioned environment and report the results in Figure 5.2. TIDBD(λ) did not increase the step-size parameter and did not converge to the optimal weight in five million steps. TD(λ) with step-size optimization, on the other hand, increased the step-size parameter until w reached

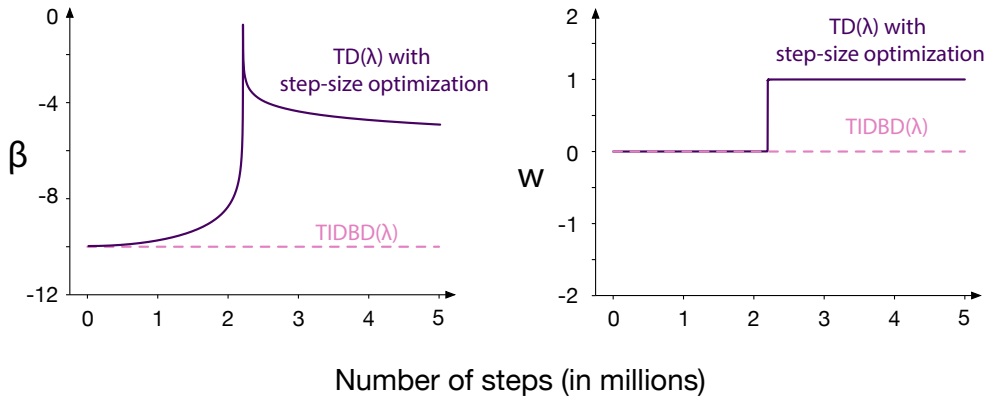

Figure 5.2: Results of TIDBD(λ) and TD(λ) with step-size optimization. TIDBD(λ) did not increase the step-size of w and as a result, did not converge to the optimal weight in five million steps. TD(λ) with step-size optimization, on the other hand, increased the step-size until w reached one. Then it slowly reduced the step-size, converging to $w = 1$.

1. Then it slowly reduced the step-size parameter, converging to $w = 1$.

The failure of TIDBD(λ) is not surprising. It is computing the meta-gradient to minimize the error w.r.t the one-step return. Since $w = 0$ already minimizes this error, the step-size parameter does not change.

5.3 The Atari Prediction Benchmark (APB)

The Atari Prediction Benchmark (APB) (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 3.1, we have to specify the observation vector, the cumulant, the discount factor γ, and the lifetime of the agent (T) for each game.

APB constructs 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. It constructs the cumulant by preprocessing the reward given by ALE. A positive reward from ALE sets the cumulant to +1, and a negative reward sets it to -1. The cumulant is zero otherwise. APB uses $\gamma = 0.98$ at all time steps. Finally, it sets the lifetime to 210,000, which is around 1 hour of gameplay at 60 frames per second.

Constructing the feature vector from the game frame

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

In the preprocessing steps, APB first resizes the frame to $105 \times 80 \times 3$. It converts each of the three channels in the resized frame to a tensor of dimensions $105 \times 80 \times 8$ by performing a binning process 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 5.3 (a) illustrates the binning process with a simple example and Figure 5.3 (b) shows the binning process applied to a frame of the game Freeway.

The binning process gives us three tensors of dimensions $105 \times 80 \times 8$. APB stacks them to get a tensor of dimensions $105 \times 80 \times 24$ and flattens the stacked tensor to get a vector with 201,600 binary components. Finally, APB appends 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 201,620 components.

5.4 Experiments: TD(λ) with Step-size Optimization on APB

I compared TD(λ) with step-size optimization with TD(λ) on the 50 games in the Atari Prediction Benchmark. For each game, I ran both algorithms for 210,000 steps with $\lambda = 0.90$ and compared the lifetime error—Equation 3.1.

An important caveat of the lifetime error is that it measures the return error, and in stochastic environments, the return error can be high even when

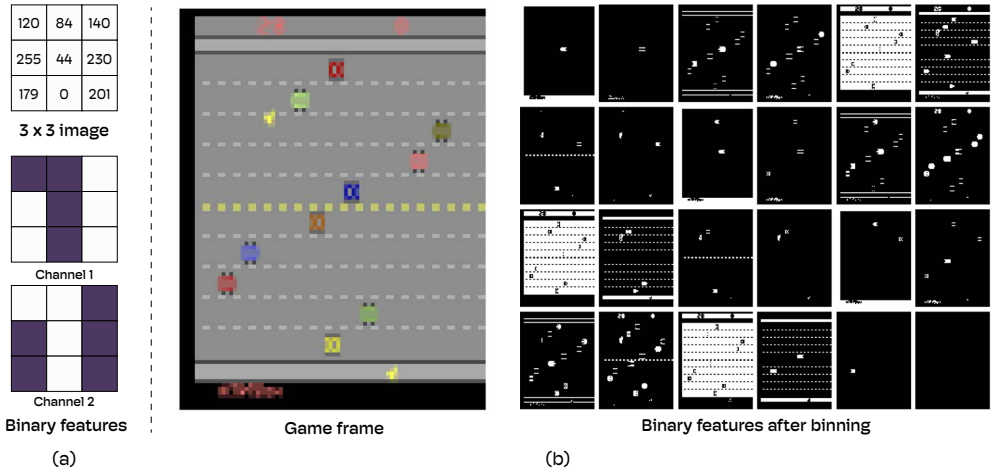

Figure 5.3: **(a)** A simplified example of the binning step with a 3×3 image. I transform the image into a binary valued tensor by binning the value of the pixel into two channels. Pixel values from 0 to 127 are binned into the first channel, and 128 to 255 into the second channel. **(b)** The binning 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.

the agent is learning well. The magnitute of the lifetime errors themselves are not meaningful. We can only use them to compare algorithms.

I plot the ratio of the lifetime error of TD(λ) with step-size optimization and the lifetime error of TD(λ). A ratio of 1 means both algorithms had the same lifetime error. A ratio of 0.5 means TD(λ) with step-size optimization had half the lifetime error of TD(λ).

I plot one set of results of learning with meta-step-size of 10^{-4} in Figure 5.4. In all direct comparisons TD(λ) with step-size optimization used α^{init} that was the same as α used by TD(λ). For all values of α^{init}, step-size optimization helped on a majority of the games. The difference was largest when α^{init} were the smallest.

I ran more experiments with a larger meta-step-size of 10^{-3} and plot the results in Figure 5.5. Once again, TD(λ) with step-size optimization achieved a lower lifetime error on many games. In some games, it diverged on a number of environments (labels shown in red). The divergence of TD(λ) with step-size optimization is concerning. If step-size optimization in only useful if the meta-step-size is carefully tuned, it might not be useful in practice.

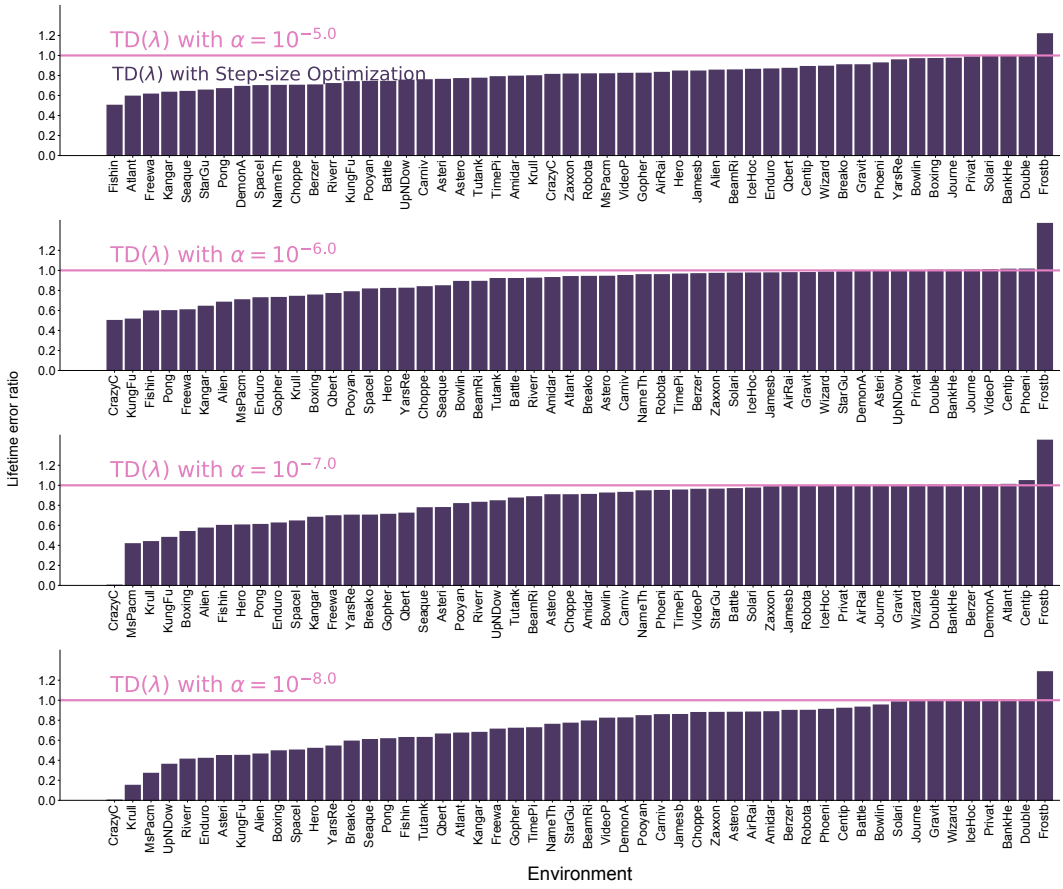

Figure 5.4: The lifetime error of TD(λ) with step-size optimization compared to the lifetime error of TD(λ). In all experiments, TD(λ) with step-size optimization used a meta-step-size of 10^{-4}. Both algorithms used $\lambda = 0.90$. In all comparisons between the two algorithms, α^{init} was the same as α.

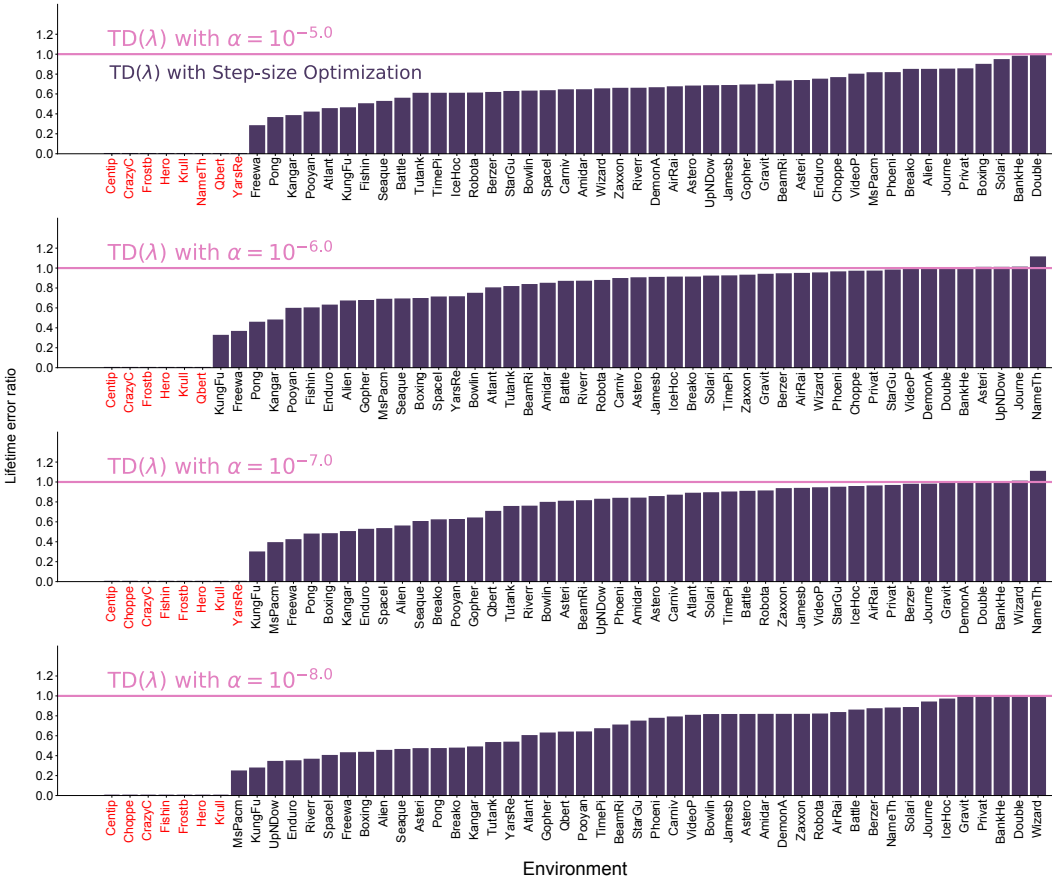

Figure 5.5: The lifetime error of TD(λ) with step-size optimization compared to the lifetime error of TD(λ). In all experiments, TD(λ) with step-size optimization used a meta-step-size of 10^{-3}. Both algorithms used $\lambda = 0.90$. In all comparisons between the two algorithms, α^{init} was the same as α. The red labels show games on which TD(λ) with step-size optimization diverged.

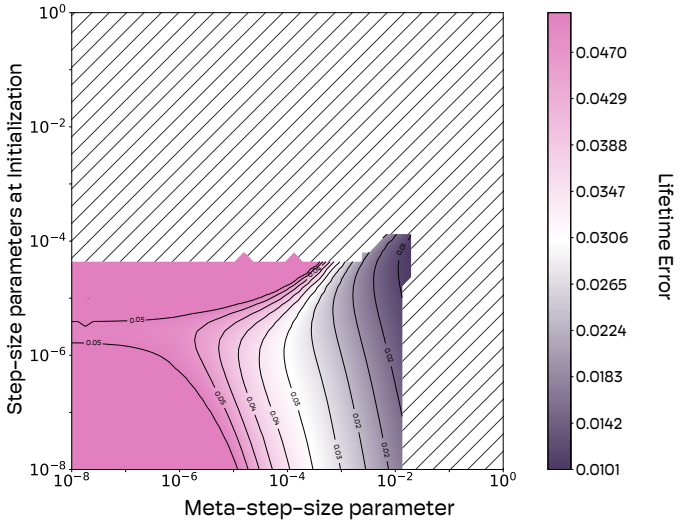

Figure 5.6: Lifetime error of TD(λ) with step-size optimization for a wide range of α^{init} and θ on Pong. The diagonal lines are hyperparameter configurations for which the algorithm diverged.

To better understand the sensitivity of TD(λ) with step-size optimization to θ and α^{init}, I ran it for fifty-five values of α^{init} and θ on the game Pong. For both parameters, I used values from the set $\{0.7^x | x \in \{0, 1, \cdots, 54\}\}$ for a total of 3025 experiments.

I plot all results in Figure 5.6, where the x-axis is the meta-step-size parameter and the y-axis is α^{init}. The lifetime error is shown with a color scale that has purple at one end (low error) and pink at the other end (high error). For some hyperparameter combinations, the lifetime error was higher than the highest value of the scale. All those values are shown as pink as well. Hyperparameters for which the agent diverged are shown by diagonal lines.

The performance of TD(λ) with step-size optimization 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 main conclusions from the sensitivity analysis is that step-size optimization improves performane for some values of the hyperparameters, and it hurts for some values.

5.5 True Online TD(λ) with Step-size Optimization

Algorithm 8: True Online TD(λ) with Step-size Optimization

Hyperparameters: $\alpha_{init}, \lambda, \theta$
Initializations: $(\boldsymbol{w}, \boldsymbol{h^{old}}, \boldsymbol{h^{temp}}, \boldsymbol{z^{\delta}}, \boldsymbol{p}, \boldsymbol{h}, \boldsymbol{z}, \bar{\boldsymbol{z}}) \leftarrow (\mathbf{0}, \cdots, \mathbf{0}), (v^{\delta}, v^{old}) = (0, 0), \boldsymbol{\beta} \leftarrow \ln(\boldsymbol{\alpha}_{init}) \in \mathbb{R}^n$
while *alive* **do**
  Receive $\boldsymbol{\phi}$, γ, and r
  $v \leftarrow \sum_{i|\phi[i]\neq 0} w[i]\phi[i]$
  $\delta' \leftarrow r + \gamma v - v^{old}$
  for $i \mid z[i] \neq 0$ **do**
    $\delta^w[i] \leftarrow \delta' z[i] - z^{\delta}[i] v^{\delta}$
    $w[i] \leftarrow w[i] + \delta^w[i]$
    $\beta[i] \leftarrow \beta[i] + \frac{\theta}{e^{\beta[i]}}(\delta' - v^{\delta})p[i]$
    $h^{old}[i] \leftarrow h[i]$
    $h[i] \leftarrow h^{temp}[i] + \delta' \bar{z}[i] - z^{\delta}[i] v^{\delta}$
    $z^{\delta}[i] = 0$
    $(z[i], p[i], \bar{z}[i]) \leftarrow (\gamma\lambda z[i], \gamma\lambda p[i], \gamma\lambda \bar{z}[i])$
  $v^{\delta} \leftarrow 0$
  $b \leftarrow \sum_{i|\phi[i]\neq 0} z[i]\phi[i]$
  for $i \mid \phi[i] \neq 0$ **do**
    $v^{\delta} \leftarrow v^{\delta} + \delta^w[i]\phi[i]$
    $z^{\delta}[i] \leftarrow e^{\beta[i]}\phi[i]$
    $z[i] \leftarrow z[i] + z^{\delta}[i](1 - b)$
    $p[i] \leftarrow p[i] + \phi[i] h^{old}[i]$
    $\bar{z}[i] \leftarrow \bar{z}[i] + z^{\delta}[i]\,(1 - b - \phi[i]\bar{z}[i])$
    $h^{temp}[i] \leftarrow h[i] - z^{\delta}[i]\phi[i]h[i] - h^{old}[i]\phi[i](z[i] - z^{\delta}[i])$
  $v^{old} \leftarrow v$

True Online TD(λ) with step-size optimization uses ${\delta'}_t$, the modified TD error, defined as:

$$ {\delta'}_t = r_t + \gamma_t v_{t-1,t} - v_{t-2,t-1}. \tag{5.18} $$

It updates the ith weight parameter as:

$$ w_t[i] = w_{t-1}[i] + {\delta'}_t z_{t-1}[i] - e^{\beta_{t-1}[i]}(v_{t-1,t-1} - v_{t-2,t-1})\phi_{t-1}[i], \tag{5.19} $$

and estimates $z_{t-1}[i]$ as:

$$ z_{t-1}[i] = \gamma_{t-1}\lambda z_{t-2}[i] + e^{\beta[i]}\phi_{t-1}[i] - e^{\beta[i]}\phi_{t-1}[i] b_{t-1}, \tag{5.20} $$

where b_{t-1} is:

$$b_{t-1} = \gamma_{t-1}\lambda \sum_{i=1}^{n} z_{t-2}[i]\phi_{t-1}[i]. \tag{5.21}$$

Let $\frac{\partial w_{t-1}[i]}{\partial \beta[i]}$ be $h_{t-1}[i]$. Then, we can approximate $h_{t-1}[i]$ incrementally as:

$$\begin{aligned}
h_{t-1}[i] &= \frac{\partial w_{t-1}[i]}{\partial \beta[i]} \\
&= \frac{\partial w_{t-2}[i]}{\partial \beta[i]} + \frac{\partial(\delta'_{t-1} z_{t-2}[i])}{\partial \beta[i]} - \phi_{t-2}[i]\frac{\partial \left(e^{\beta[i]}(v_{t-2,t-2} - v_{t-3,t-2})\right)}{\partial \beta[i]} \\
&= h_{t-2}[i] + z_{t-2}[i]\frac{\partial \delta'_{t-1}}{\partial \beta[i]} + \delta'_{t-1}\frac{\partial z_{t-2}[i]}{\partial \beta[i]} - \phi_{t-2}[i]e^{\beta[i]}\frac{\partial(v_{t-2,t-2} - v_{t-3,t-2})}{\partial \beta[i]} \\
&- \phi_{t-2}[i](v_{t-2,t-2} - v_{t-3,t-2})e^{\beta[i]}.
\end{aligned} \tag{5.22}$$

Using a similar approximation of the meta-gradient as done by IDBD, we simply the above equation as:

$$\begin{aligned}
h_{t-1}[i] \approx\ & h_{t-2}[i] + z_{t-2}[i]\frac{\partial \delta'_{t-1}}{\partial \beta[i]} + \delta'_{t-1}\frac{\partial z_{t-2}[i]}{\partial \beta[i]} - \phi_{t-2}[i]e^{\beta[i]}(h_{t-2}[i]\phi_{t-2}[i] - h_{t-3}[i]\phi_{t-2}[i]) \\
&- \phi_{t-2}[i](v_{t-2,t-2} - v_{t-3,t-2})e^{\beta[i]}.
\end{aligned} \tag{5.23}$$

The gradient $\frac{\partial \delta'_{t-1}}{\partial \beta[i]}$ is approximated as:

$$\begin{aligned}
\frac{\partial \delta'_{t-1}}{\partial \beta[i]} &= \frac{\partial(r_{t-1} + \gamma_{t-1}\sum_{j=1}^{n} w_{t-2}[j]\phi_{t-1}[j] - \sum_{j=1}^{n} w_{t-3}[j]\phi_{t-2}[j])}{\partial \beta[i]} \\
&\approx -h_{t-3}[i]\phi_{t-2}[i].
\end{aligned} \tag{5.24}$$

Finally, if $\bar{z}_{t-2}[i]$ is $\frac{\partial z_{t-2}[i]}{\partial \beta[i]}$, then it can be recursively approximated as:

$$\begin{aligned}
\bar{z}_{t-2}[i] &= \frac{\partial}{\partial \beta[i]}\left(\gamma_t \lambda z_{t-3} + e^{\beta[i]}\phi_{t-2}[i] - e^{\beta[i]}\phi_{t-2}[i]b_{t-2}\right) \\
&= \gamma_{t-2}\lambda \bar{z}_{t-3}[i] + e^{\beta[i]}\phi_{t-2}[i] - e^{\beta[i]}\phi_{t-2}[i]b_{t-2} - e^{\beta[i]}\phi_{t-2}[i]\frac{\partial b_{t-2}}{\partial \beta[i]} \\
&\approx \gamma_{t-2}\lambda \bar{z}_{t-3}[i] + e^{\beta[i]}\phi_{t-2}[i] - e^{\beta[i]}\phi_{t-2}[i]b_{t-2} - \gamma_{t-2}\lambda e^{\beta[i]}\phi_{t-2}[i]^2 \bar{z}_{t-3}[i] \\
&= \gamma_{t-2}\lambda \bar{z}_{t-3}[i] + e^{\beta[i]}\phi_{t-2}[i]\left(1 - b_{t-2} - \gamma_{t-2}\lambda \phi_{t-2}[i]\bar{z}_{t-3}[i]\right).
\end{aligned} \tag{5.25}$$

Combining the above equations, $h_{t-1}[i]$ is approximated as:

$$
\begin{aligned}
h_{t-1}[i] \approx & h_{t-2}[i] + z_{t-2}[i]\,(-h_{t-3}[i]\phi_{t-2}[i]) \\
& + \delta'_{t-1}\bar{z}_{t-2}[i] - \phi_{t-2}[i]e^{\beta[i]}(h_{t-2}[i]\phi_{t-2}[i] - h_{t-3}[i]\phi_{t-2}[i]) \\
& - \phi_{t-2}[i](v_{t-2,t-2} - v_{t-3,t-2})e^{\beta[i]} \\
= & h_{t-2}[i] + z_{t-2}[i]\,(-h_{t-3}[i]\phi_{t-2}[i]) \\
& + \delta'_{t-1}\bar{z}_{t-2}[i] - \phi_{t-2}[i]^2 e^{\beta[i]}(h_{t-2}[i] - h_{t-3}[i]) \\
& - \phi_{t-2}[i](v_{t-2,t-2} - v_{t-3,t-2})e^{\beta[i]} \\
= & h_{t-2}[i] - h_{t-3}[i]\phi_{t-2}[i]\left(z_{t-2}[i] - e^{\beta[i]}\phi_{t-2}[i]\right) \\
& + \delta'_{t-1}\bar{z}_{t-2}[i] - \phi_{t-2}[i]e^{\beta[i]}h_{t-2}[i]\phi_{t-2}[i] \\
& - \phi_{t-2}[i](v_{t-2,t-2} - v_{t-3,t-2})e^{\beta[i]} \\
= & h_{t-2}[i] - h_{t-3}[i]\phi_{t-2}[i]\left(z_{t-2}[i] - e^{\beta[i]}\phi_{t-2}[i]\right) \\
& + \delta'_{t-1}\bar{z}_{t-2}[i] - \phi_{t-2}[i]e^{\beta[i]}(h_{t-2}[i]\phi_{t-2}[i] \\
& + (v_{t-2,t-2} - v_{t-3,t-2})).
\end{aligned}
\tag{5.26}
$$

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 Algorithm 8.

5.6 Experiments: True Online TD(λ) with Step-size Optimization on APB

Similar to earlier experiments, I evaluated the performance of True Online TD(λ) with step-size optimization on the Atari Prediction Benchmark.

Figure 5.7 and Figure 5.8 show the performance of True Online TD(λ) with step-size optimization compared to True Online TD(λ) for different values of the meta-step-size parameter and the initial step-size parameter. The initial step-size parameter in all comparisons is the same as the step-size parameter of the baseline. The results are similar to the results of TD(λ) with step-size optimization. The performance of True Online TD(λ) with step-size optimization improved on a majority of the games. It diverged on some games.

Figure 5.9 is the hyperparameter sensitivity analysis of True Online TD(λ)

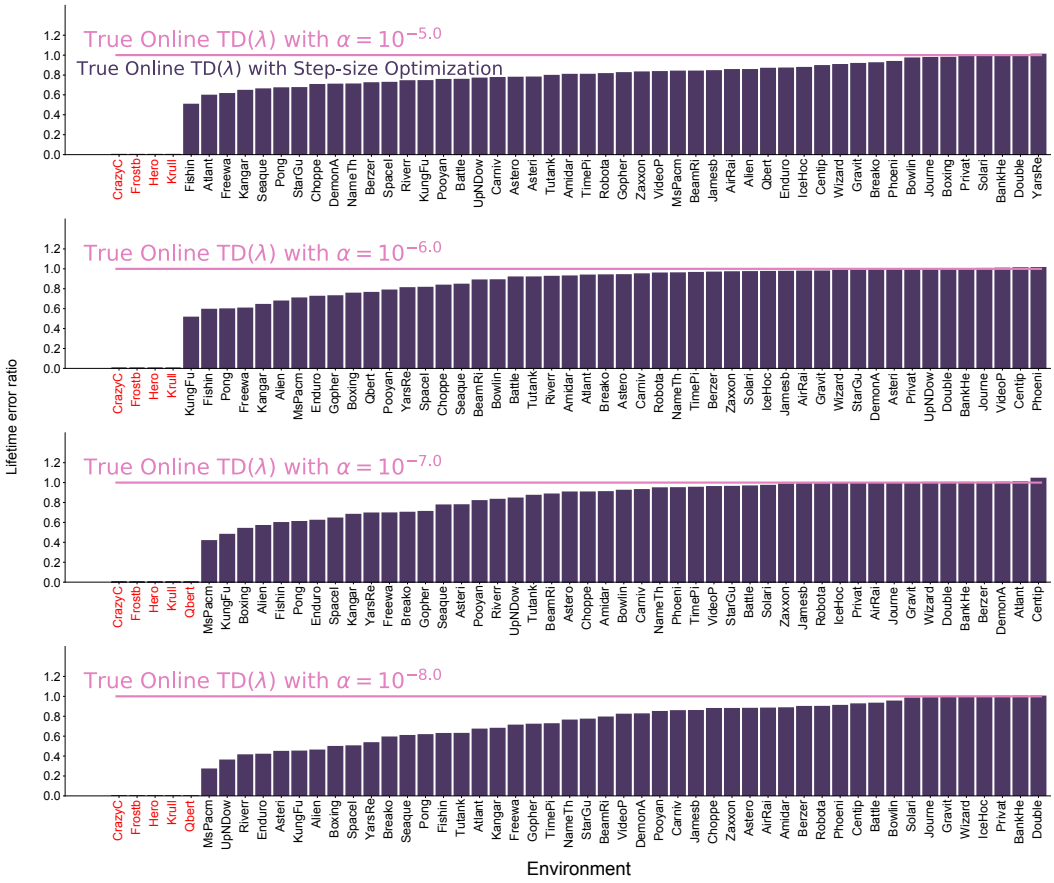

Figure 5.7: The lifetime error of True Online TD(λ) with step-size optimization compared to the lifetime error of True Online TD(λ). In all experiments, True Online TD(λ) with step-size optimization used a meta-step-size of 10^{-4}. In all comparisons between the two algorithms, α^{init} was the same as α. The red labels show games on which True Online TD(λ) with step-size optimization diverged.

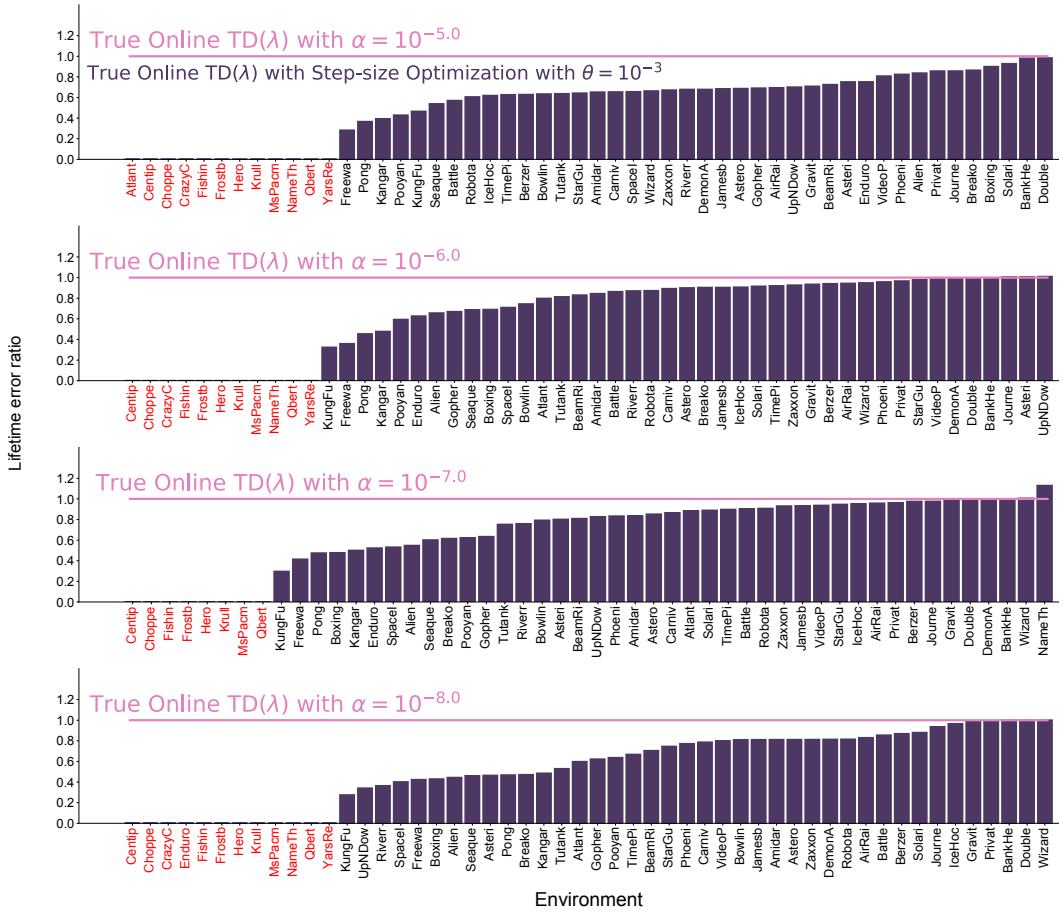

Figure 5.8: The lifetime error of True Online TD(λ) with step-size optimization compared to the lifetime error of True Online TD(λ). In all experiments, True Online TD(λ) with step-size optimization used a meta-step-size of 10^{-3}. In all comparisons between the two algorithms, α^{init} was the same as α. The red labels show games on which True Online TD(λ) with step-size optimization diverged.

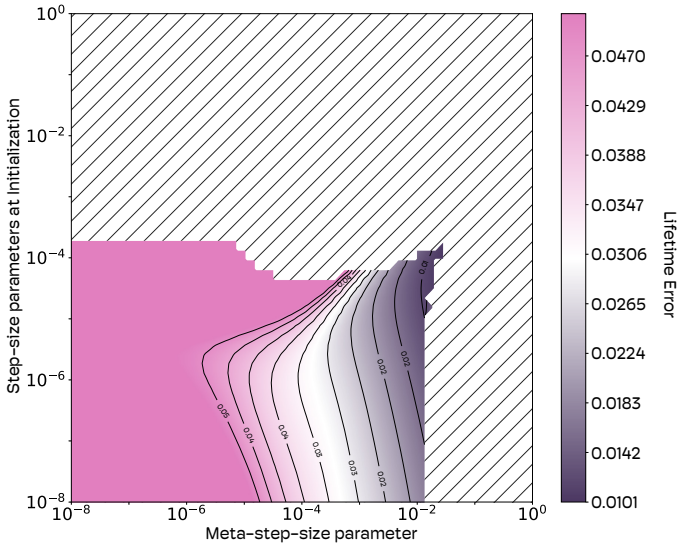

Figure 5.9: Lifetime error of True Online TD(λ) with step-size optimization for a wide range of α^{init} and θ on Pong. The diagonal lines are hyperparameter configurations for which the algorithm diverged.

with step-size optimization on Pong. The performance of the algorithm improved 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 results are similar to the results of TD(λ) with step-size optimization.

Comparing TD(λ) with Step-size Optimization and True Online TD(λ) with Step-size Optimization

I compared the performance of TD(λ) with step-size optimization and True Online TD(λ) with step-size optimization on three games—Bowling, Atlantis, and Seaquest—and plot their performance as a function of the meta-step-size parameter and the initial step-size parameter in Figure 5.10. The best performance of both algorithms was comparable, and they both diverged for similar values of their hyperparameters.

The results on APB show that step-size optimization can improve performance of TD learning algorithms for some values of their hyperparameters. The results do not show any advantage of True Online TD(λ) over TD(λ) when they are combined with step-size optimization. In the next chapter, I

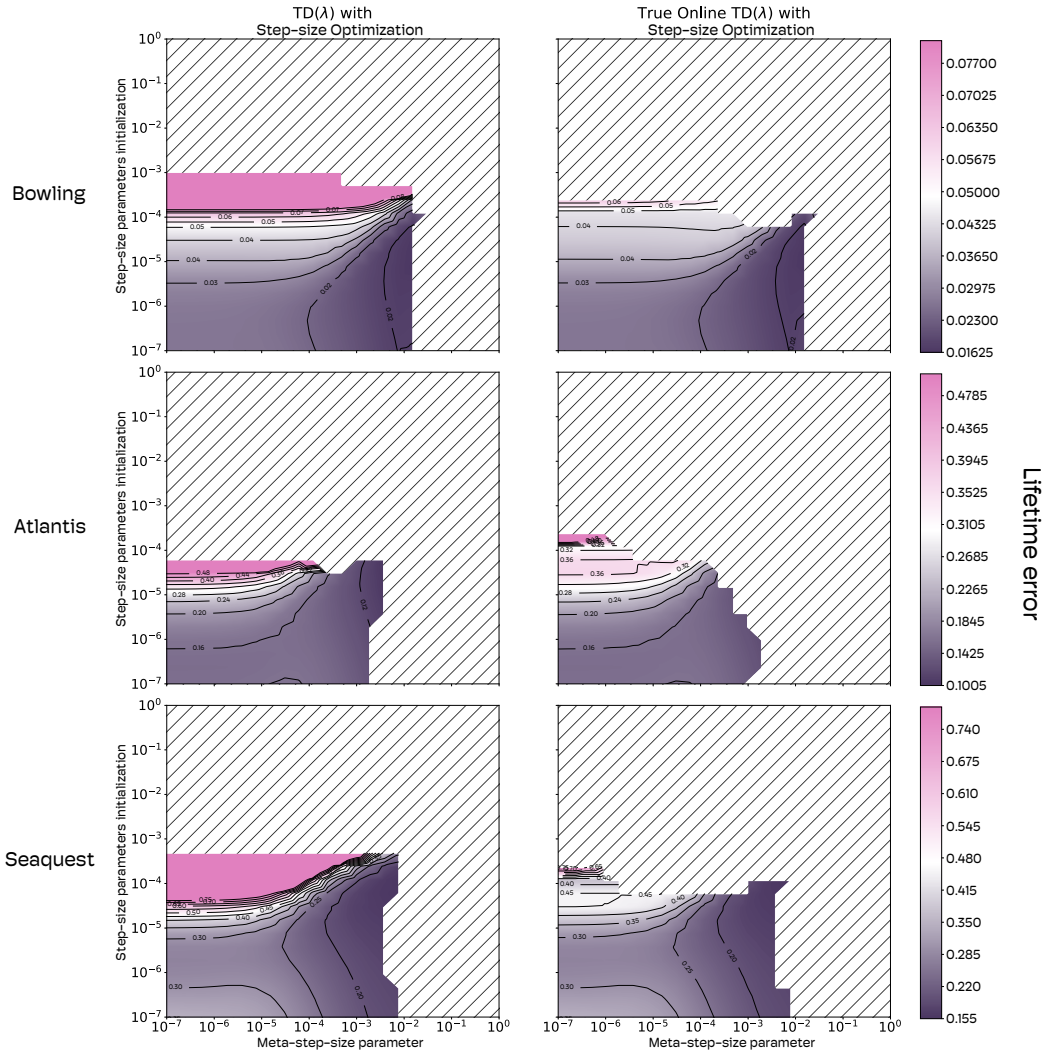

Figure 5.10: The performance of TD(λ) with step-size optimization (first column) and True Online TD(λ) with step-size optimization (second column) for a wide range of meta-step-size parameters and initial step-size parameters. The rows are results on different games. The diagonal lines are hyperparameters for which the algorithms diverged. The best performance of both algorithms was comparable, and they both diverged for similar values of their hyperparameters. The added complexity of True Online TD(λ) did not provide any advantage over TD(λ) when they were combined with step-size optimization.

present an algorithm that fixes the issue of divergence of True Online TD(λ) and True Online TD(λ) with step-size optimzation.

Chapter 6

Temporal Difference Learning with the Overshoot Bound

In the last chapter I showed that per-feature step-size parameters and step-size optimization can be combined with TD(λ) and True Online TD(λ) to improve their performance. Step-size optimization works well if the initial value of the step-size parameters and the meta-step-size parameter are tuned properly. Otherwise, it can cause algorithms to diverge. In this chapter I develop a bound on the update to the weight parameters that fixes the issue of divergence. The idea behind the bound is to limit the magnitude of the update such that the update never makes the error on a sample worse than it was before the update. At the same time, the bound is not too conservative, and it does not hinder quick learning.

6.1 Correction Ratio of a Learning Update

I define the *correction ratio* of a learning update as the fraction of the prediction error reduced after the update on the sample used in the update. Let y^*_t be the target of a learning system at time step t. y^*_t could be a ground-truth target, as in supervised learning, or a bootstrapped target, such as a λ-return. If $\boldsymbol{\phi}_t$ and $\boldsymbol{w}_{t-1}$ are the feature vector and the weight parameter vector at time step t, then the prediction error before the update is:

$$l_t = y^*_t - \sum_{i=1}^{n} w_{t-1}[i]\phi_t[i], \tag{6.1}$$

and the prediction error after the update is:

$$l'_t = y^*_t - \sum_{i=1}^{n} w_t[i]\phi_t[i]. \tag{6.2}$$

The correction ratio, τ_t, for this update is defined as:

$$\tau_t \doteq \frac{l_t - l'_t}{l_t}. \tag{6.3}$$

We can develop some intuition about the correction ratio with some examples. Consider the case when the target is y^*_t and the prediction is $0.5y^*_t$ before the update. If the prediction after the update is $0.75y^*_t$—the prediction error is half of what it was before—the correction ratio would be 0.5. If the prediction after the update is $1.25y^*_t$—the error is half in magnitude but opposite in sign—the correction ratio would be 1.5. A correction ratio of 1 would make the prediction perfectly match the target after the update. A correction ratio larger than 2 or less 0 would increase the error after the update, for example, a correction ratio of three or negative one would mean that the magnitude of the error after the update is twice as large as before.

If we know the update rule for the weight parameters, then we can derive a simpler expression for τ_t. Consider the case when the weights are updated using stochastic gradient descent, and the gradient is computed w.r.t the squared prediction error. Let $\boldsymbol{\alpha}_t$ be the step-size parameter vector. The update rule for the ith weight parameter for this learning system is:

$$w_t[i] = w_{t-1}[i] + \alpha_t[i]\phi_t[i]l_t, \tag{6.4}$$

and the correction ratio is:

$$
\begin{aligned}
\tau_t &= \frac{(y_t^* - \sum_{i=1}^{n} w_t[i]\phi_t[i]) - (y_t^* - \sum_{i=1}^{n} w_{t+1}[i]\phi_t[i])}{(y_t^* - \sum_{i=1}^{n} w_t[i]\phi_t[i])} \\
\implies \tau_t l_t &= (y_t^* - \sum_{i=1}^{n} w_t[i]\phi_t[i]) - (y_t^* - \sum_{i=1}^{n} w_{t+1}[i]\phi_t[i]) \\
\tau_t l_t &= -\sum_{i=1}^{n} w_t[i]\phi_t[i] + \sum_{i=1}^{n}(w_t[i] + \alpha_t[i] l_t \phi[i])\phi_t[i] \\
\tau_t l_t &= \sum_{i=1}^{n} \alpha_t[i] l_t \phi[i]^2 \\
\tau_t l_t &= l_t \sum_{i=1}^{n} \alpha_t[i] \phi[i]^2 \\
\tau_t &= \sum_{i=1}^{n} \alpha_t[i] \phi[i]^2.
\end{aligned}
\tag{6.5}
$$

The correction ratio is a function of the step-size parameter vector and the feature vector. In the linear case the feature vector cannot be changed by the agent, and the only way to control the correction ratio is to adjust the step-size parameter vector. A bound can be designed that adapts the step-size parameter vector to ensure that the correction ratio of every update reduces the prediction error. I call this bound *the overshoot bound*.

6.2 Overshoot Bound for Linear Regression

If the ground-truth target for a given sample is y^* and the agent's prediction is $0.5y^*$, then the overshoot bound guarantees that the prediction after the update is between $0.5y^*$ and y^*. [1]

We can formalize this intuition with an inequality for prediction error on a sample before and after the update. If

$$
\text{Case 1: } y_t^* - \sum_{i=1}^{n} w_t[i]\phi_t[i] \geq 0 \text{ and } y_t^* - \sum_{i=1}^{n} w_{t-1}[i]\phi_t[i] \geq 0, \tag{6.6}
$$

[1] A more general constraint is that the prediction after the update should be between $0.5y^*$ and $1.5y^*$ but overshooting the ground-truth target makes little sense to me, even if the prediction error is reduced.

then:

$$
\begin{aligned}
& y_t^* - \sum_{i=1}^{n} w_{t-1}[i]\phi_t[i] \geq y_t^* - \sum_{i=1}^{n} w_t[i]\phi_t[i] \geq 0 \\
\implies & \sum_{i=1}^{n} w_{t-1}[i]\phi_t[i] \leq \sum_{i=1}^{n} w_t[i]\phi_t[i] \leq y \\
\implies & \sum_{i=1}^{n} w_{t-1}[i]\phi_t[i] \leq \sum_{i=1}^{n} \left(w_{t-1}[i] + \alpha_t[i]\phi_t[i]l\right)\phi_t[i] \leq y \\
\implies & 0 \leq \sum_{i=1}^{n} \left(\alpha_t[i]\phi_t[i]^2 l\right) \leq y - \sum_{i=1}^{n} w_{t-1}[i]\phi_t[i] \\
\implies & 0 \leq l \sum_{i=1}^{n} \alpha_t[i]\phi_t[i]^2 \leq l \\
\implies & 0 \leq \sum_{i=1}^{n} \alpha_t[i]\phi_t[i]^2 \leq 1.
\end{aligned}
\tag{6.7}
$$

Similarly, if:

$$
\text{Case 2: } y_t^* - \sum_{i=1}^{n} w_t[i]\phi_t[i] \leq 0 \text{ and } y_t^* - \sum_{i=1}^{n} w_{t-1}[i]\phi_t[i] \leq 0, \tag{6.8}
$$

then:

$$
\begin{aligned}
& 0 \leq -y_t^* + \sum_{i=1}^{n} w_{t-1}[i]\phi_t[i] \leq -y_t^* + \sum_{i=1}^{n} w_t[i]\phi_t[i] \\
\implies & 0 \leq \sum_{i=1}^{n} \alpha_t[i]\phi_t[i]^2 \leq 1.
\end{aligned}
\tag{6.9}
$$

If $y_t^* - \sum_{i=1}^{n} w_{t-1}[i]\phi_t[i]$ and $y_t^* - \sum_{i=1}^{n} w_t[i]\phi_t[i]$ have opposite signs, then the prediction overshoots the target after the update. We can ignore this case because it is not desirable.

Forcing $0 \leq \tau \leq 1$ for every update guarantees that learning updates do not increase prediction errors on samples used to perform the updates, and the predictions after the updates do not overshoot the ground-truth targets.

The overshoot bound can be implemented without introducing new hyper-parameters by setting the step-size parameter at every step to $\min(\frac{\alpha_t[i]}{\sum_{i=1}^{n} \alpha_t[i]_t\phi_t[i]^2}, \alpha_t[i])$. Algorithm 9 implements the bound for linear regression.

The overshoot bound for linear regression is not new. AutoStep (Mahmood, 2012) used a similar bound to make linear regression robust. I extend it to TD learning.

Algorithm 9: Linear Regression with the Overshoot Bound

Initializations: $\boldsymbol{w} \leftarrow \mathbf{0} \in \mathbb{R}^n$
while *alive* **do**
  Receive $\boldsymbol{\phi}$, $\boldsymbol{\alpha}$, and y^*
  $l \leftarrow y^* - \sum_{i=1}^n w[i]\phi[i]$
  $\tau_t \leftarrow \sum_{i=1}^n \alpha_t[i]\phi[i]^2$
  for $i \mid \phi[i] \neq 0$ **do**
    $w[i] \leftarrow w[i] + \min(1, \frac{1}{\tau_t})\alpha_t[i]\phi[i]l$

6.3 Overshoot Bound for TD Learning

Extending the overshoot bound to TD learning poses two challenges. The bootstrapped targets used in TD learning depend on the weight paramaters, and the targets in TD learning can be delayed by many steps.

The naive way to extend it for TD learning is to repeat the analysis we did for linear regression for TD(λ). The ith weight parameter in TD(λ) is updated as:

$$w_t[i] = w_{t-1}[i] + \alpha_t[i] z_{t-1}[i] \delta_t, \tag{6.10}$$

where $\boldsymbol{z}_{t-1}$ is the eligibility trace vector whose components are updated as:

$$z_t[i] = \gamma\lambda z_{t-1}[i] + \phi_t[i]. \tag{6.11}$$

The target used in a single update is the TD error:

$$G_{t:t+1} = r_{t+1} + \gamma \sum_{i=1}^n w_t[i]\phi_{t+1}[i], \tag{6.12}$$

which depends on the weight parameter vector. Let $G'_{t:t+1}$ be the target with the updated weight parameter vector, that is,

$$\begin{aligned} G'_{t:t+1} &= r_{t+1} + \gamma \sum_{i=1}^n w_{t+1}[i]\phi_{t+1}[i] \\ &= r_{t+1} + \gamma \sum_{i=1}^n (w_t[i] + \alpha_t[i] z_t[i] \delta_t)\phi_{t+1}[i] \\ &= r_{t+1} + \gamma \sum_{i=1}^n w_t[i]\phi_{t+1}[i] + \gamma \sum_{i=1}^n \alpha_t[i] z_t[i] \delta_t \phi_{t+1}[i] \\ &= G_{t:t+1} + \gamma\delta_t \sum_{i=1}^n \alpha_t[i] z_t[i] \phi_{t+1}[i]. \end{aligned} \tag{6.13}$$

Using the TD error as the prediction error, and the one-step bootstrapped return as the target, we get the following expression of the correction ratio:

$$
\begin{aligned}
\tau_t &= \frac{(G_{t:t+1} - \sum_{i=1}^{n} w_t[i]\phi_t[i]) - \left(G'_{t:t+1} - \sum_{i=1}^{n} w_{t+1}[i]\phi_t[i]\right)}{\delta_t} \\
\implies \tau_t\delta_t &= (G_{t:t+1} - \sum_{i=1}^{n} w_t[i]\phi_t[i]) - (G'_{t:t+1} - \sum_{i=1}^{n} w_{t+1}[i]\phi_t[i]) \\
\tau_t\delta_t &= -\sum_{i=1}^{n} w_t[i]\phi_t[i] + \sum_{i=1}^{n}(w_t[i] + \alpha_t[i]\delta_t z_t[i])\phi_t[i] - \gamma\alpha_t[i]\delta_t \sum_{i=1}^{n} z_t[i]\phi_{t+1}[i] \\
\tau_t\delta_t &= \sum_{i=1}^{n} \alpha_t[i]\delta_t z_t[i]\phi_t[i] - \gamma\alpha_t[i]\delta_t \sum_{i=1}^{n} z_t[i]\phi_{t+1}[i] \\
\tau_t\delta_t &= \delta_t \sum_{i=1}^{n} \alpha_t[i] z_t[i](\phi_t[i] - \gamma\phi_{t+1}[i]) \\
\tau_t &= \sum_{i=1}^{n} \alpha_t[i] z_t[i](\phi_t[i] - \gamma\phi_{t+1}[i]).
\end{aligned}
\tag{6.14}
$$

We can repeat the analysis with inequalities we used for linear regression to get an overshoot bound for TD learning. The bound is:

$$
0 \leq \sum_{i=1}^{n} \alpha_t[i] z_t[i](\phi_t[i] - \gamma\phi_{t+1}[i]) \leq 1. \tag{6.15}
$$

Equation 6.15 was first derived by Dabney & Barto (2012). We can use it as an overshoot bound for TD learning. Let's call the algorithm that uses this overshoot bound *TD(λ) with α-bound* (see Algorithm 18 for the pseudocode).

TD(λ) with α-bound is naive in two ways. First, it uses the one-step return as the target when estimating the prediction error instead of the λ-return target. The one-step target is not the objective of TD(λ). Second, it takes into account the change in the target after the update when TD learning ignores the influence of the weight parameters on the target.

Taking into account the influence of the update on the target can make the bound too lenient. Consider the case when the feature vectors for time step t and $t+1$ are identical, and $\gamma = 1$. Then τ_t is

$$
\sum_{i=1}^{n} \alpha_t[i] z_t[i](\phi_t[i] - \gamma\phi_{t+1}[i]) = 0. \tag{6.16}
$$

For this case, α-bound is satisfied for any values of the step-size parameters and any change in the weight parameter vector, irrespective of the magnitude.

A better way to define the correction ratio for TD learning is to use the λ-return as the target and assume that the target stays the same after the update. Consider a hypothetical learner that updates its ith weight parameter as:

$$w_t[i] = w_{t-1}[i] + \alpha_t[i]\phi_{t-1}\left(G_t^\lambda - \sum_{i=1}^{n} w_{t-1}[i]\phi_{t-1}[i]\right). \tag{6.17}$$

If we ignore the influence of the update on the target, then the correction ratio of the above update is:

$$\begin{aligned} \tau_t &= \frac{\left(G_t^\lambda - \sum_{i=1}^{n} w_t[i]\phi_t[i]\right) - \left(G_t^\lambda - \sum_{i=1}^{n} w_{t+1}[i]\phi_t[i]\right)}{\left(G_t^\lambda - \sum_{i=1}^{n} w_t[i]\phi_t[i]\right)} \\ \tau_t &= \sum_{i=1}^{n} \alpha_t[i]\phi_t[i]^2, \end{aligned} \tag{6.18}$$

which is the same as the correction ratio for linear regression. The overshoot bound for this hypothetical learner is:

$$0 \leq \sum_{i=1}^{n} \alpha_t[i]\phi_t[i]^2 \leq 1. \tag{6.19}$$

Using the bound in Equation 6.19 is not straightforward because no practical algorithm uses Equation 6.17 as its update rule.

We know that the weight updates done by True Online TD(λ) eventually add up to the update in Equation 6.17 after many steps. Fortunately, True Online TD(λ) with time-dependent step-size parameter, an algorithm we discussed in Chapter 2, provides a practical way to implement the bound in Equation 6.19. It updates the components of its eligibility trace vector as:

$$z_t[i] = \gamma\lambda z_{t-1}[i] + \alpha[i]\phi_t[i](1-b), \tag{6.20}$$

where b is:

$$b = \lambda\gamma_t \sum_{i=1}^{n} \alpha_t[i]\phi_t[i]z_{t-1}[i]. \tag{6.21}$$

A modified update to the eligibility trace vector of True Online TD(λ) can incorporate the overshoot bound. This update is:

$$z_t[i] = \gamma\lambda z_{t-1}[i] + \min\left(1, \frac{1}{\tau_t}\right)\alpha[i]\phi_t[i]. \tag{6.22}$$

Algorithm 10: True Online TD(λ) with the Overshoot Bound

Hyperparameters: $\boldsymbol{\alpha}, \lambda$
Initializations: $\boldsymbol{w}, \boldsymbol{z^\delta}, \boldsymbol{z} \leftarrow \mathbf{0} \in \mathbb{R}^n; (v^\delta, v^{old}) = (0, 0)$
while *alive* **do**
  Receive $\boldsymbol{\phi}$, γ, and r
  $v \leftarrow \sum_{\phi[i] \neq 0} w[i]\phi[i]$
  $\delta' \leftarrow r + \gamma v - v^{old}$
  for $i \mid z[i] \neq 0$ **do**
    $\delta^w[i] \leftarrow \delta' z[i] - z^\delta[i] v^\delta$
    $w[i] \leftarrow w[i] + \delta^w[i]\ z^\delta[i] = 0$
    $z[i] \leftarrow \gamma\lambda z[i]$
  $v^\delta \leftarrow 0$
  $\tau_t \leftarrow \sum_{i=0}^{n} \alpha_t[i]\phi[i]^2$
  $b \leftarrow \sum_{\phi[i] \neq 0} z[i]\phi[i]$
  for $i \mid \phi[i] \neq 0$ **do**
    $v^\delta \leftarrow v^\delta + \delta^w[i]\phi[i]$
    $z^\delta[i] \leftarrow \min(1, \frac{1}{\tau_t})\alpha_t[i]\phi[i]$
    $z[i] \leftarrow z[i] + z^\delta[i](1 - b)$
  $v^{old} \leftarrow v$

I call the algorithm that uses this modified update rule *True Online TD(λ) with the overshoot bound* (see Algorithm 10 for the pseudocode).

We can derive an analogous algorithm for TD(λ). The updates done by TD(λ) do not add up to the update in Equation 6.17. We can still apply a similar modification to the update rule of the eligibility trace vector of TD(λ) by changing:

$$z_t[i] = \gamma\lambda z_{t-1}[i] + \alpha[i]\phi_t[i], \tag{6.23}$$

to:

$$z_t[i] = \gamma\lambda z_{t-1}[i] + \min(1, \frac{1}{\tau_t})\alpha[i]\phi_t[i]. \tag{6.24}$$

I call the algorithm that uses this modified update rule *TD(λ) with the overshoot bound* (see Algorithm 11 for the pseudocode). Note that *TD(λ) with the overshoot bound* only approximately satisfies the bound in Equation 6.19.

Algorithm 11: TD(λ) with the Overshoot Bound

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

6.4 Experiments: TD learning with the Overshoot Bound on APB

I empirically compare TD(λ) with the overshoot bound to True Online TD(λ) with the overshoot bound by running both algorithms on the Atari Prediction Benchmark for a wide range of step-size parameter vectors. All components of the step-size parameter vector are set to the same value.

Ideally, we should expect the performance of the algorithms to improve as the step-size parameters increase. Eventually, the performance might get worse if the step-size parameter is too large. Once the bound is triggered, any further increase in the step-size parameters should not impact performance.

Figure 6.1, 6.2, and 6.3 plot the results on all games. TD(λ) with the overshoot bound diverged in some games for large step-size parameters. In some games, it did not diverge but had very high lifetime error. The bound, when applied to TD(λ), did not solve the instability of learning with large step-size parameters. The results for True Online TD(λ) with the bound tell a different story. In all games the bound fixed the instability of learning with large step-size parameters.

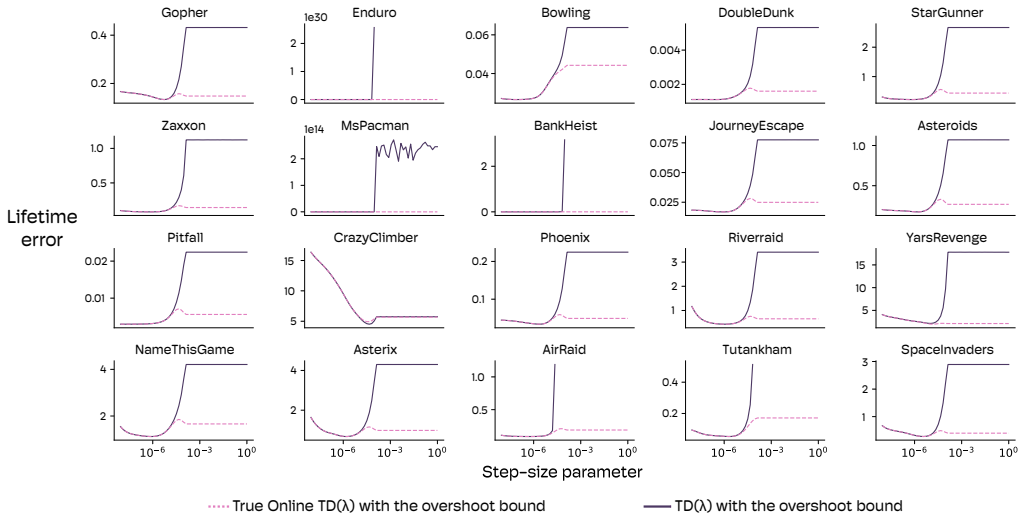

Figure 6.1: Comparing TD(λ) with the overshoot bound and True Online TD(λ) with the overshoot bound. The latter fixes the instability of learning with large step-size parameters, and the former does not.

6.5 Overshoot Bound for True Online TD(λ) with Step-size Optimization

Similar to True Online TD(λ), True Online TD(λ) with step-size optimization can be modified to incorporate the overshoot bound to get *True Online TD(λ) with step-size optimization and the overshoot bound* (See Algorithm 10 for the pseudocode). Note that in the pseudocode, the traces of the meta-gradients are reset whenever the bound is triggered. This is because the hard scaling of the step-size parameters when applying the bound is not differentiable, and the meta-gradients are not well-defined. Setting the meta-gradients to zero is a naive way to handle this issue.

I compare this new algorithm with True Online TD(λ) with step-size optimization on the game of Pong. The results are in Figure 6.4. The overshoot bound fixed the instability of learning with large step-size parameters, and t algorithm with the bound did not diverge for any values of the meta-step-size parameter and initial step-size parameter. Moreover, the performance of the algorithm with the bound did not change for hyperparameters for which the algorithm without the bound did not diverge. This highlights that the bound is not overly conservative, and it only plays a role when needed.

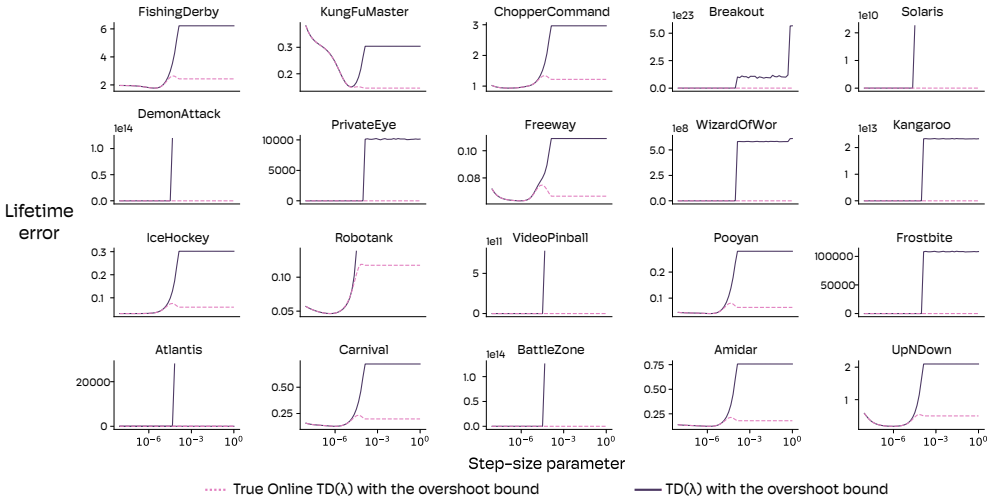

Figure 6.2: Comparing TD(λ) with the overshoot bound and True Online TD(λ) with the overshoot bound. The latter fixes the instability of learning with large step-size parameters, and the former does not.

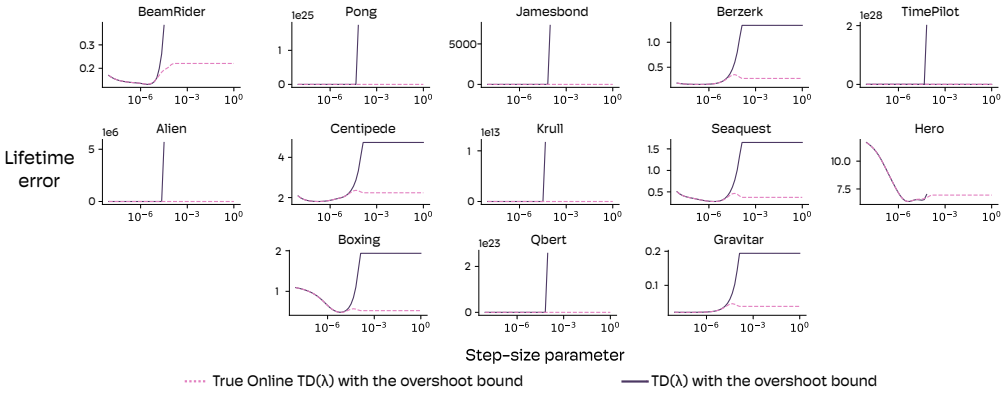

Figure 6.3: Comparing TD(λ) with the overshoot bound and True Online TD(λ) with the overshoot bound. The latter fixes the instability of learning with large step-size parameters, and the former does not.

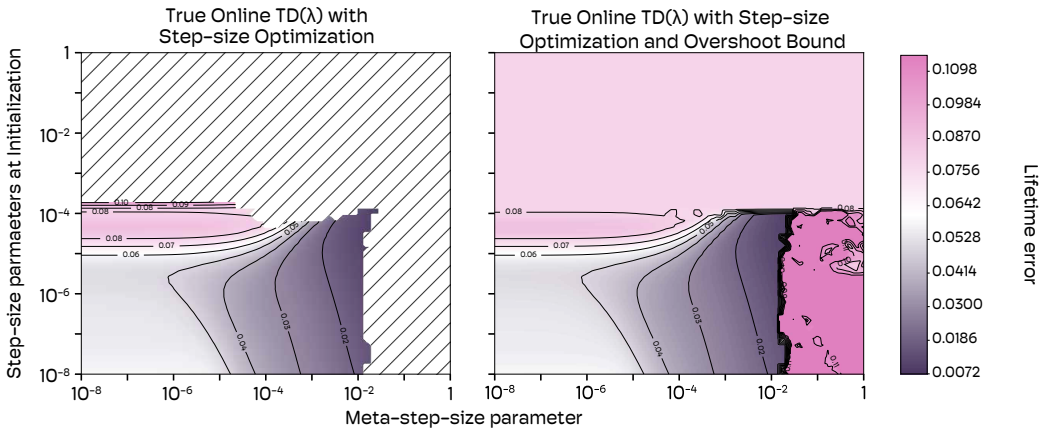

Figure 6.4: True Online TD(λ) with step-size optimization compared to True Online TD(λ) with step-size optimization and the overshoot bound on the game of Freeway. The latter does not diverge for any values of the meta-step-size parameter and initial step-size parameter, showing the effectiveness of the bound.

The overshoot bound, when applied to True Online TD(λ) with step-size optimization, can prevent divergence for large step-size parameters and meta-step-size parameters. Still, the performance for large step-size parameters and meta-step-size parameters is not good. In the next chapter, I introduce a mechanism for reducing the step-size parameters when they are too large.

Algorithm 12: True Online TD(λ) with Step-size Optimization and Overshoot Bound

Hyperparameters: $\alpha^{init}, \lambda, \theta$

Initializations: $\boldsymbol{w}, \boldsymbol{h^{old}}, \boldsymbol{h^{temp}}, \boldsymbol{z^{\delta}}, \boldsymbol{p}, \boldsymbol{h}, \boldsymbol{z}, \bar{\boldsymbol{z}} \leftarrow \mathbf{0} \in \mathbb{R}^n; (v^{\delta}, v^{old}) = (0,0); \beta[i] \leftarrow \alpha^{init}\ \forall\ i$

while *alive* **do**

  Perceive $\boldsymbol{\phi}, \gamma$ and r

  $v \leftarrow \sum_{\phi[i]\neq 0} w[i]\phi[i]$

  $\delta' \leftarrow r + \gamma v - v^{old}$

  for $i \mid z[i] \neq 0$ **do**

    $\delta^w[i] \leftarrow \delta' z[i] - z^{\delta}[i] v^{\delta}$

    $w[i] \leftarrow w[i] + \delta^w[i]$

    $\beta[i] \leftarrow \beta[i] + \frac{\theta}{e^{\beta[i]}}(\delta' - v^{\delta})p[i]$

    $h^{old}[i] \leftarrow h[i]$

    $h[i] \leftarrow h^{temp}[i] + \delta' \bar{z}[i] - z^{\delta}[i] v^{\delta}$

    $z^{\delta}[i] = 0$

    $(z[i], p[i], \bar{z}[i]) \leftarrow (\gamma\lambda z[i], \gamma\lambda p[i], \gamma\lambda \bar{z}[i])$

  $v^{\delta} \leftarrow 0$

  $\tau_t \leftarrow \sum_{i \mid \phi[i]\neq 0} e^{\beta[i]} \phi[i]^2$

  $b \leftarrow \sum_{\phi[i]\neq 0} z[i]\phi[i]$

  for $i \mid \phi[i] \neq 0$ **do**

    $v^{\delta} \leftarrow v^{\delta} + \delta^w[i]\phi[i]$

    $z^{\delta}[i] \leftarrow \min\left(1, \frac{1}{\tau_t}\right) e^{\beta[i]}\phi[i]$ `// Overshoot bound`

    $z[i] \leftarrow z[i] + z^{\delta}[i](1-b)$

    $p[i] \leftarrow p[i] + \phi[i] h^{old}[i]$

    $\bar{z}[i] \leftarrow \bar{z}[i] + z^{\delta}[i]\,(1 - b - \phi[i]\bar{z}[i])$

    $h^{temp}[i] \leftarrow h[i] - z^{\delta}[i]\phi[i]h[i] - h^{old}[i]\phi[i](z[i] - z^{\delta}[i])$

    if $\tau > 1$ **then**

      $(h^{temp}[i], h[i], \bar{z}[i]) = (0,0,0)$

  $v^{old} \leftarrow v$

Chapter 7

SwiftTD: Fast and Robust Temporal Difference Learning

In the last two chapters, I showed that step-size optimization improved the performance of TD learning, and the overshoot bound made TD learning robust to poorly chosen step-size parameters and meta-step-size parameters. In this chapter, I introduce *SwiftTD*, a TD learning algorithm that combines the ideas of step-size optimization and *η-bound* (a generalization of the overshoot bound) with a third idea called *step-size decay*.

7.1 True Online TD(λ) with the η-bound

The overshoot bound is a powerful idea that prevents divergence in TD learning by scaling updates just enough so that the updated predictions do not overshoot the targets. Whenever the bound is triggered, the predictions are updated to exactly match the targets. If the observations or targets in a problem are noisy, matching the targets can be too large of an update, and it can be beneficial to be more restrictive. The *η-bound* provides a way to be more restrictive.

It generalizes the overshoot bound by restricting the overcorrection ratios of updates to be below η, a hyperparameter that is between 0 and 1. Recall that the overcorrection ratio for True Online TD(λ) using $\boldsymbol{\alpha}$ as the step-size parameter vector is:

$$\tau_t = \sum_{i=1}^{n} \phi_t[i]\alpha_t[i]^2, \tag{7.1}$$

and the eligibility trace vector is updated as:

$$z_t[i] = \lambda\gamma_t z_{t-1}[i] + \alpha_t[i]\phi_t[i](1-b), \tag{7.2}$$

where b is:

$$b = \lambda\gamma_t \sum_{i=1}^{n} z_{t-1}[i]\phi_t[i]. \tag{7.3}$$

True Online TD(λ) with η-bound updates its eligibility trace vector as:

$$z_t[i] = \min(1, \frac{\eta}{\tau_t})\alpha_t[i]\phi_t[i]. \tag{7.4}$$

Note that $\eta = 1$ makes the η-bound identical to the overshoot bound.

7.2 Step-size Decay

The η-bound scales down the step-size parameters temporarily and has no lasting impact on the values of the step-size parameters. Step-size optimization can reduce the step-size parameters if they are too large, but only if the update rule is differentiable. The η-bound, when used, makes the learning update non-differentiable, and step-size optimization can no longer effectively reduce the step-size parameters. The idea behind step-size decay is simple: if the η-bound is triggered, then we reduce the step-size parameters by a small value. Reducing them is sensible because the bound is only triggered when they are too large.

Mechanistically step-size decay is simple to implement. Let $\boldsymbol{\alpha}_t$ be the step-size parameter vector at time step t. At every step for which the η-bound is triggered, the ith step-size parameter is updated as:

$$\alpha_{t+1}[i] = \alpha_t[i]\epsilon^{\phi_t[i]^2}, \tag{7.5}$$

where ϵ is a hyperparameter called the decay rate. A reasonable choice for ϵ is a value close to one, such as 0.99 or 0.999. Pseudocode for True Online TD(λ) with step-size decay is Algorithm 13.

Related work

The idea of step-size decay is similar in spirit to the *step-size ratchet* algorithm by Ghiassian (2022). It differs from step-size ratchet in three important ways.

Algorithm 13: True Online TD(λ) with Step-size Decay

Hyperparameters: $\eta = 0.5, \alpha^{init} = 10^{-7}, \epsilon = 0.99, \lambda, \gamma, \theta$
Initializations: $\boldsymbol{w}, \boldsymbol{z}^{\boldsymbol{\delta}}, \boldsymbol{z} \leftarrow \mathbf{0} \in \mathbb{R}^n; (v^{\delta}, v^{old}) = (0, 0)$
while *alive* **do**
  Receive $\boldsymbol{\phi}$, γ, and r
  $v \leftarrow \sum_{i|\phi[i]\neq 0} w[i]\phi[i]$
  $\delta' \leftarrow r + \gamma v - v^{old}$
  for $i \mid z[i] \neq 0$ **do**
    $\delta^w[i] \leftarrow \delta' z[i] - z^{\delta}[i] v^{\delta}$
    $w[i] \leftarrow w[i] + \delta^w[i]\ z^{\delta}[i] = 0$
    $z[i] \leftarrow \gamma\lambda z[i]$
  $v^{\delta} \leftarrow 0$
  $\tau \leftarrow \sum_{i|\phi[i]\neq 0} \alpha[i]\phi[i]^2$
  $b \leftarrow \sum_{i|\phi[i]\neq 0} z[i]\phi[i]$
  for $i \mid \phi[i] \neq 0$ **do**
    $v^{\delta} \leftarrow v^{\delta} + \delta^w[i]\phi[i]$
    $z^{\delta}[i] \leftarrow \min(1, \frac{\eta}{\tau})\alpha\phi[i]$
    $z[i] \leftarrow z[i] + z^{\delta}[i](1 - b)$
    if $\tau > \eta$ **then**
      $\alpha[i] \leftarrow \alpha[i]\epsilon^{\phi[i]^2}$
  $v^{old} \leftarrow v$

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, unlike step-size decay, it does not decay step-size parameters proportional to their contribution to the overcorrection ratio. Instead, step-size parameters of all features, even if the features are zero, are reduced.

7.3 SwiftTD: Fast and Robust TD Learning

SwiftTD combines step-size optimization, the η-bound, step-size decay, and True Online TD(λ) in a single algorithm. In addition, it clips the step-size parameters to be in range $[e^{\eta_{min}}, e^{\eta}]$ at every time step, where η_{min} is a hyper-parameter.

The pseudocode for SwiftTD is Algorithm 14. The pink lines implement

step-size optimization, the blue lines implement the η-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).

Algorithm 14: SwiftTD

Hyperparameters: $\epsilon = 0.999, \eta = 0.1, \eta^{min} = e^{-15}, \alpha^{init} = 10^{-7}, \lambda, \theta$
Initializations: $\boldsymbol{w}, \boldsymbol{h^{old}}, \boldsymbol{h^{temp}}, \boldsymbol{z^{\delta}}, \boldsymbol{p}, \boldsymbol{h}, \boldsymbol{z}, \bar{\boldsymbol{z}} \leftarrow \mathbf{0} \in \mathbb{R}^n; (v^{\delta}, v^{old}) = (0,0). \boldsymbol{\beta} \leftarrow \ln(\boldsymbol{\alpha}^{init}) \in \mathbb{R}^n$
while *alive* **do**
  Perceive $\boldsymbol{\phi}$ and r
  $v \leftarrow \sum_{i|\phi[i]\neq 0} w[i]\phi[i]$
  $\delta' \leftarrow r + \gamma v - v^{old}$
  for $i \mid z[i] \neq 0$ **do**
    $\delta^w[i] \leftarrow \delta' z[i] - z^{\delta}[i] v^{\delta}$
    $w[i] \leftarrow w[i] + \delta^w[i]$
    $\beta[i] \leftarrow \beta[i] + \frac{\theta}{e^{\beta[i]}}(\delta' - v^{\delta})p[i]$
    $\beta[i] \leftarrow \text{clip}\,(\beta[i], \ln(\eta^{min}), \ln(\eta))$
    $h^{old}[i] \leftarrow h[i]$
    $h[i] \leftarrow h^{temp}[i] + \delta' \bar{z}[i] - z^{\delta}[i] v^{\delta}$
    $z^{\delta}[i] = 0$
    $(z[i], p[i], \bar{z}[i]) \leftarrow (\gamma\lambda z[i], \gamma\lambda p[i], \gamma\lambda \bar{z}[i])$
  $v^{\delta} \leftarrow 0$
  $\tau \leftarrow \sum_{i|\phi[i]\neq 0} e^{\beta[i]} \phi[i]^2$
  $b \leftarrow \sum_{i|\phi[i]\neq 0} z[i]\phi[i]$
  for $i \mid \phi[i] \neq 0$ **do**
    $v^{\delta} \leftarrow v^{\delta} + \delta^w[i]\phi[i]$
    $z^{\delta}[i] \leftarrow \min\left(1, \frac{\eta}{\tau}\right) e^{\beta[i]}\phi[i]$ `// The η-bound`
    $z[i] \leftarrow z[i] + z^{\delta}[i](1 - b)$
    $p[i] \leftarrow p[i] + \phi[i] h^{old}[i]$
    $\bar{z}[i] \leftarrow \bar{z}[i] + z^{\delta}[i]\,(1 - b - \phi[i]\bar{z}[i])$
    $h^{temp}[i] \leftarrow h[i] - z^{\delta}[i]\phi[i]h[i] - h^{old}[i]\phi[i](z[i] - z^{\delta}[i])$
    if $\tau > \eta$ **then**
      $\beta[i] = \beta[i] + \phi[i]^2 \ln(\epsilon)$ `// Step-size decay`
      $(h^{temp}[i], h[i], \bar{z}[i]) = (0, 0, 0)$
  $v^{old} \leftarrow v$

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 η-bound to prevent bad updates while simultaneously

reducing them proportional to their contribution to the correction ratios.

7.4 Experiments: SwiftTD on the Atari Prediction Benchmark

To demonstrate the effectiveness of SwiftTD I compare it to True Online TD(λ) with step-size optimization and True Online TD(λ) with step-size optimization and the overshoot bound on the game of Pong for a wide range of meta-step-size parameters and initial step-size parameters.

I plot the results in Figure 7.1. I used $\eta = 0.1$ and $\epsilon = 0.999$ for SwifTD. SwiftTD performed well for almost all combinations of α^{init} and θ. True Online TD(λ) with step-size optimization and the overshoot bound, on the other hand, performed poorly when the initial value of the step-size parameters or the meta-step-size parameter were too large.

I then compared SwiftTD and True Online TD(λ) on all Atari games. For both SwiftTD and True Online TD(λ), I swept over all their hyperparameters. Because SwiftTD has more hyperparameters, I did a coarser search over its hyperparameters for a fair comparison. The details of the hyperparameter sweeps are in Appendix B.2.

I tuned all hyperparameters for each Atari game individually and used 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. I verified that the results did not change qualitatively with either choice.

I plot individual learning curves for eight games in Figure 7.3. In each plot, the y-axis is the lifetime error and the x-axis is the lifetime. In each of the eight games, SwiftTD had a smaller lifetime error for almost all lifetime parameters.

I plot the predictions made by both methods in the final 3,000 steps on four games in Figure 7.2. The gray dotted lines are the return from each time step. Predictions learned by SwiftTD were significantly more accurate. In

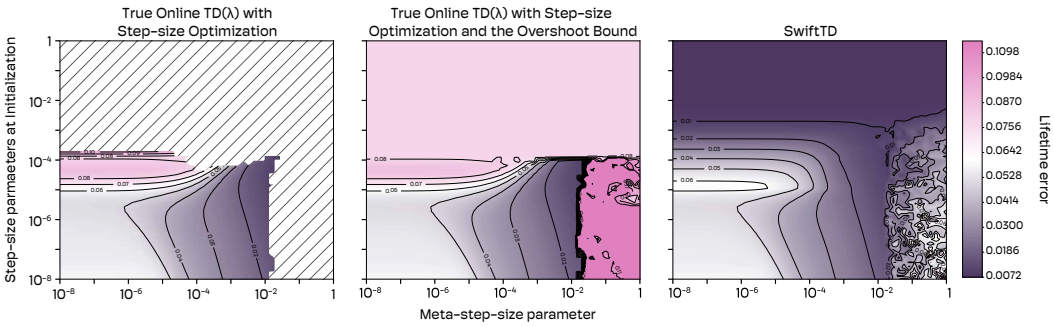

Figure 7.1: Parameter sensitivity study of SwiftTD and baselines. I ran SwiftTD, True Online TD(λ) with step-size optimization, and SwiftTD without step-size decay of 55 values of α_{init} and meta-step-size parameter for a total of 3025 experiments each. I then plot the prediction error

some games—Altantis, Pooyan—True Online TD(λ) completely failed for all hyperparameter settings whereas SwiftTD learned accurate predictions.

I also compared the performance of SwiftTD with fixed hyperparameters of $\eta = 0.1$, $\epsilon = 0.999$, $\theta = 3^{-3}$ and $\alpha^{init} = 10^{-6}$ to True Online TD(λ) with different values of α. The results are in Figure 7.4. SwiftTD performed as well or better on all games using the same hyperparameters.

7.5 Experiments: Hyperparameter Sensitivity Study of SwiftTD

In another set of experiments I studied the sensitivity of SwiftTD to its hyperparameters ϵ, η, θ, and α^{init}. I ran SwiftTD on three games: Altantis, SpaceInvaders, and Sequest for 25 values of α^{init} and θ, four values of ϵ, and four values of η for a total of 10,000 experiments on each game. I plot the results in Figures 7.5, 7.6, and 7.7.

In all three games, step-size decay improved performance for large meta-step-size parameters and large initial step-size parameters. A value of 0.999 performed well, whereas a smaller value of 0.9 degraded performance. The η-bound also improved performance, with $\eta = 0.03$ performing better than $\eta = 0.3$ on all three games. $\eta = 0.01$ performed slightly worse than $\eta = 0.03$.

The hyperparameter sensitivity provides evidence that the best hyperparameters do not vary widely across games and can be set reliably. Step-size

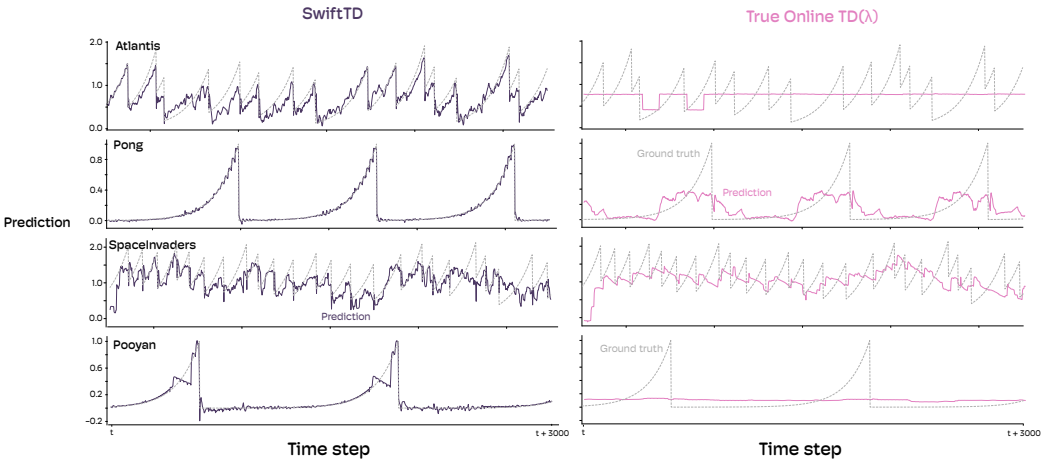

Figure 7.2: Predictions made by True Online TD(λ) and SwiftTD after learning for two hours of gameplay on Atari games. The gray dotted lines show the ground-truth returns. SwiftTD learned significantly more accurate predictions than True Online TD(λ). In some games—Pong, Pooyan—the predictions were near perfect. Even in more difficult games, like SpaceInvaders, the predictions anticipated the onset rewards.

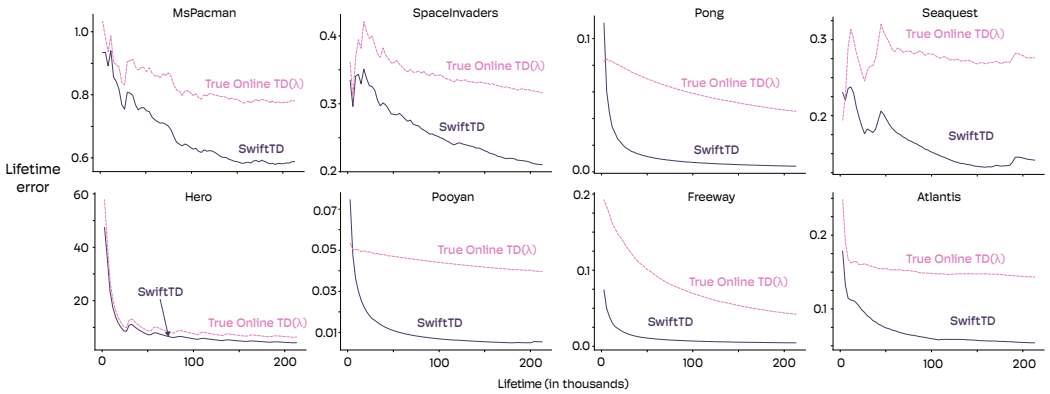

Figure 7.3: Learning curves for eight games. The y-axis is $\mathcal{L}$(time step). In all games, SwiftTD reduced error faster than True Online TD(λ). Note that because we are plotting the return error, the minimum achievable error would not be zero in stochastic environments such as Atari. The minimum error cannot be estimated from experience. Consequently, the y-axis should only be used to compare algorithms and not to measure absolute performance.

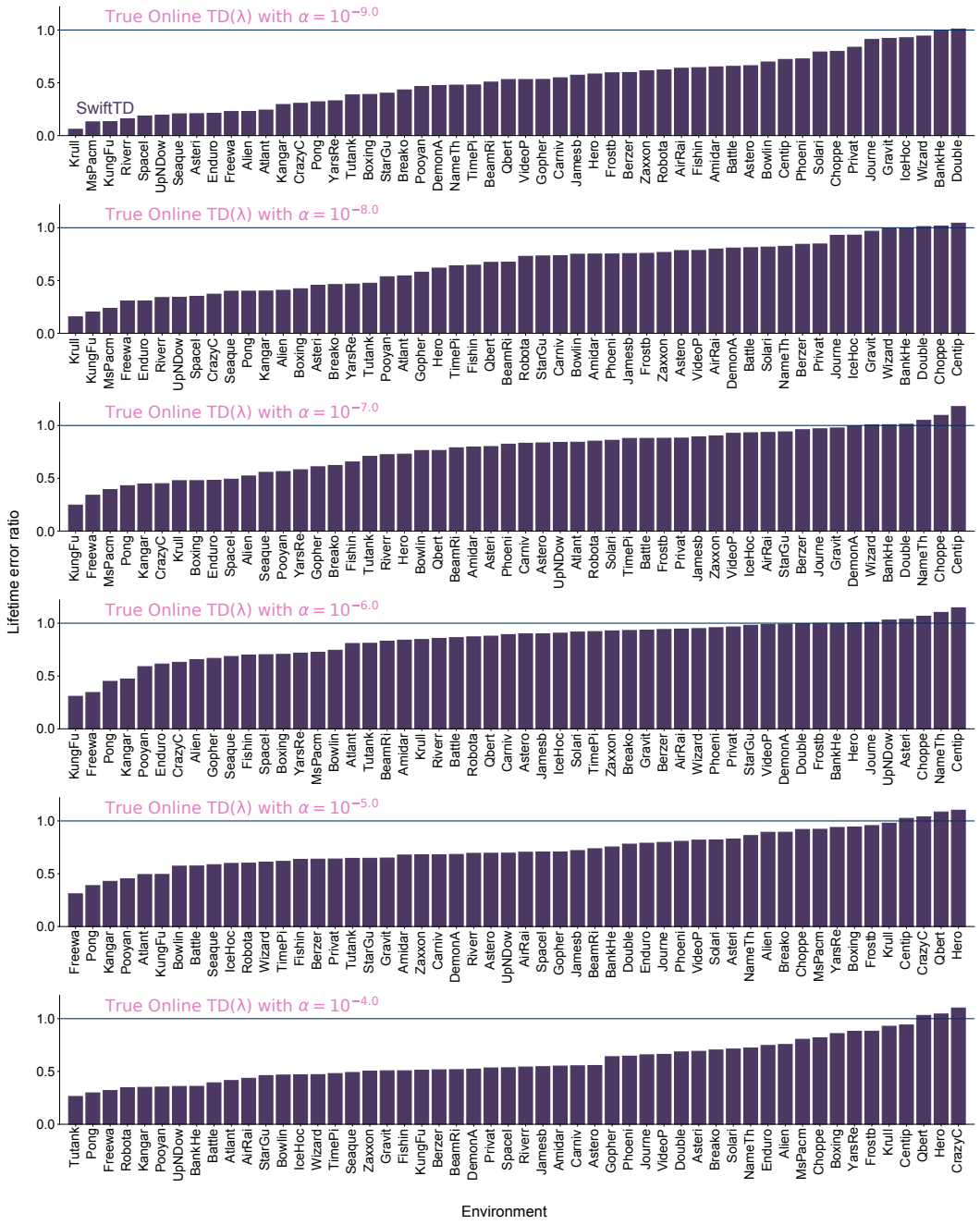

Figure 7.4: SwiftTD with fixed hyperparameters compared to True Online TD(λ) with different values of the step-size parameter. For all values of α, SwiftTD achieved a lower lifetime error than True Online TD(λ) on a majority of the games.

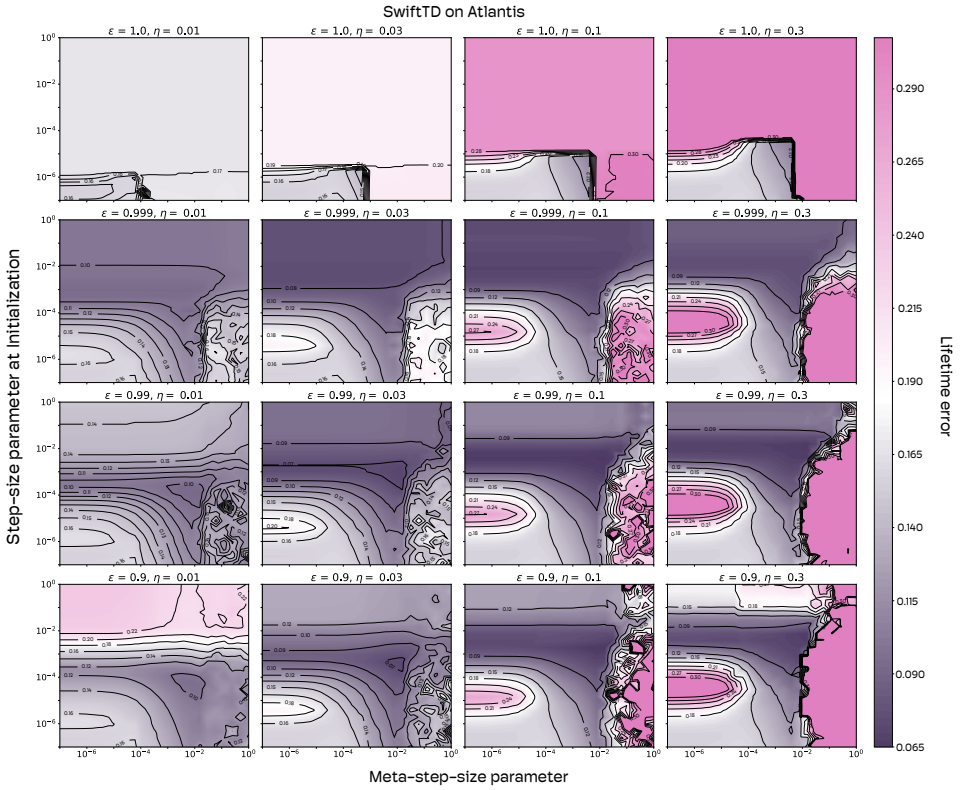

Figure 7.5: Hyperparameter sensitivity study of SwiftTD on the game Atlantis. Comparing the plots for $\epsilon = 1$ and $\epsilon = 0.999$, we see that step-size decay improved the performance for large meta-step-size and large initializations of the step-size parameters. Using a more restrictive bound also improved performance as $\eta = 0.03$ performed better than $\eta = 0.3$.

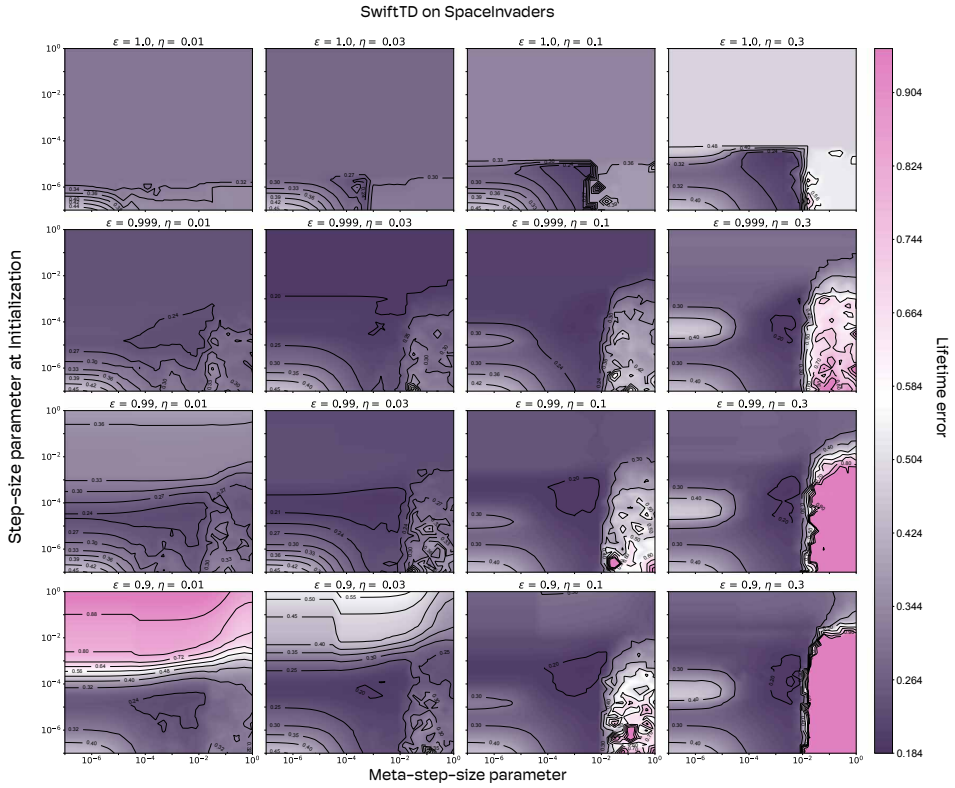

Figure 7.6: Hyperparameter sensitivity study of SwiftTD on the game SpaceInvaders. Comparing the plots for $\epsilon = 1$ and $\epsilon = 0.999$, we see that step-size decay improved the performance for large meta-step-size and large initializations of the step-size parameters. Using a more restrictive bound also improved performance as $\eta = 0.03$ performed better than $\eta = 0.3$.

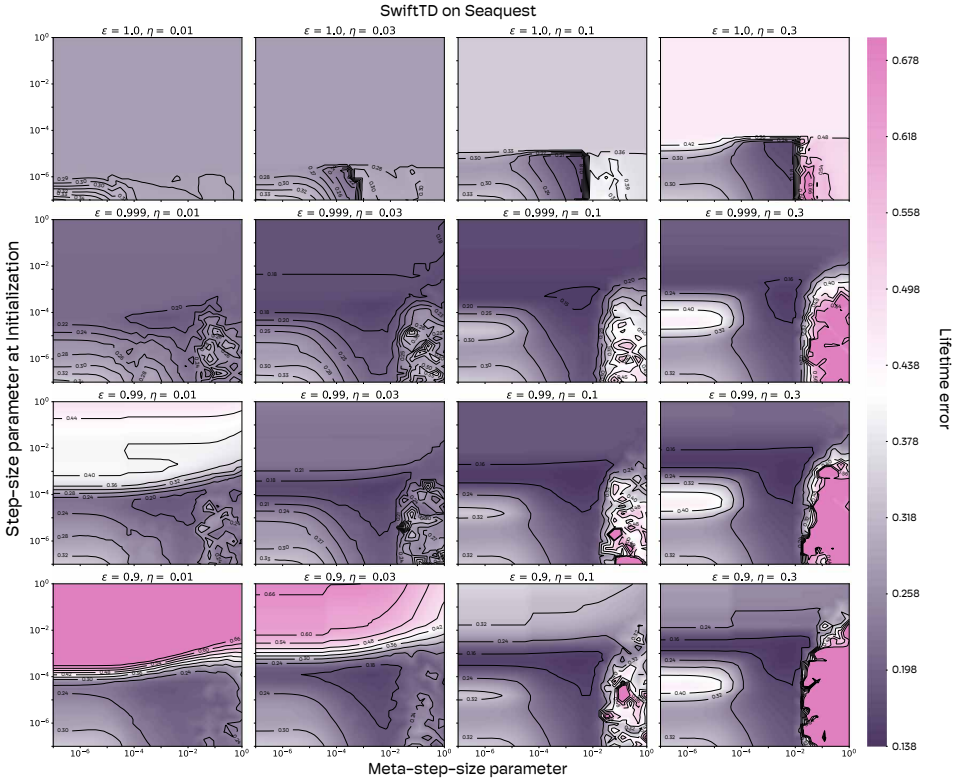

Figure 7.7: Hyperparameter sensitivity study of SwiftTD on the game Sequest. Comparing the plots for $\epsilon = 1$ and $\epsilon = 0.999$, we see that step-size decay improved the performance for large meta-step-size and large initializations of the step-size parameters. Using a more restrictive bound also improved performance as $\eta = 0.03$ performed better than $\eta = 0.3$.

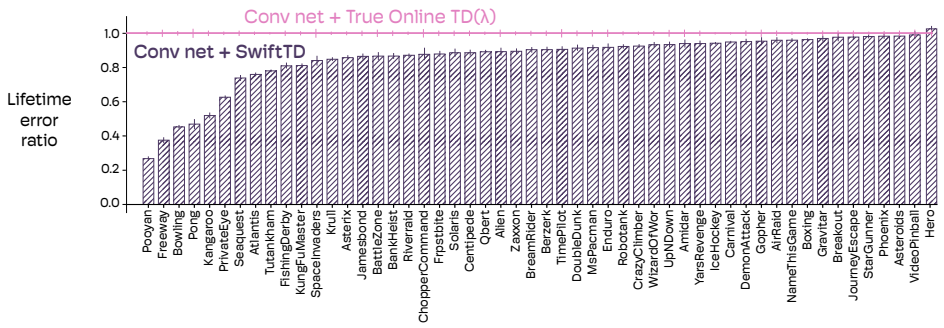

Figure 7.8: Comparing performance of convolutional networks on the Atari Prediction Benchmark. SwiftTD significantly outperformed True Online TD(λ) even when combined with neural networks. The confidence intervals are $+-$ two standard error around the mean computed over fifteen runs.

decay of 0.999 and $\eta = 0.1$ or 0.03 are reasonable default values.

7.6 Experiments: SwiftTD with Convolutional Neural Networks

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

Instead of using the preprocessing described in Chapter 5, I used SwiftTD with a one-layer convolutional neural network. I applied a convolutional layer on the $105 \times 80 \times 24$ tensor I got after stacking the three tensors given by the binning process described in Section 5.3. The convolutional layer had 25 kernels of size $3 \times 3 \times 24$ each. The weights of the kernels were initialized by sampling from $\mathcal{U}(-1, 1)$.

I 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, and it is passed through the ReLU activation function (Fukushima, 1969) and flattened to get a feature vector with 52,000 components. A weight parameter vector is used to make linear predictions from the feature vector. The main challenge in applying SwiftTD to neural networks was that SwiftTD was developed for linear learners. I 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 the baseline, I used True Online TD(λ) in the last layer.

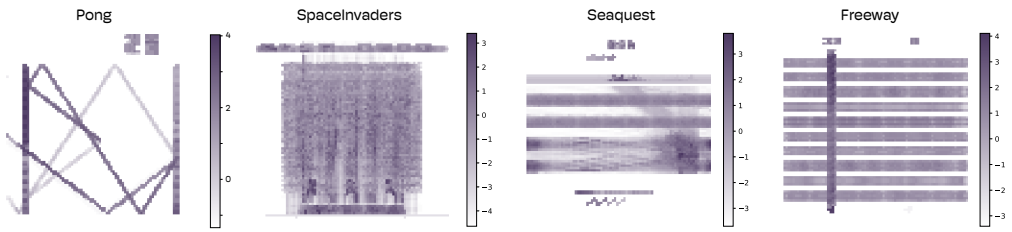

Figure 7.9: Visualizing the amount of credit assigned to each pixel by SwiftTD over the lifetime of the agent. The color map is in the log space. We see that SwiftTD assigned credit to meaningful aspects of the game. For example, in Pong, it assigned credit to the trajectories of the ball. In MsPacman, 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.

I 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 7.8. 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.

7.7 Experiments: Credit Assignment by SwiftTD

The motivation behind step-size optimization was to selectively increase the step-size parameters of features that are predictive of the return. We can visualize how well SwiftTD achieved this goal by initializing step-size parameters to small values and visualizing the amount of credit assigned to each pixel of the game frames over the lifetime of the agent.

On any given step, I define the credit assigned to the ith feature by SwiftTD as the quantity added to its eligibility trace, that is,

$$\min(1, \frac{\eta}{\tau_t}) e^{\beta_t[i]} \phi_t[i]^2. \tag{7.6}$$

The average credit assigned to the ith feature over the lifetime of the agent is:

$$\text{Credit}^T[i] = \frac{1}{T} \sum_{t=1}^{T} \min(1, \frac{\eta}{\tau_t}) e^{\beta_t[i]} \phi_t[i]^2. \tag{7.7}$$

If we run an experiment where the initial value of the step-size parameters is very small, then $\text{Credit}^T[i]$ measures the learned importance of the ith feature by SwiftTD.

I ran SwiftTD on eight games with an initial step-size parameter of 10^{-8} and measured $\mathbf{Credit}^{\boldsymbol{T}}$ at the end of learning. Recall that our APB learners have 201,619 features. The last 19 features encode the actions and the cumulant, and the remaining 201,600 features are the binned pixels of the game frame. $\mathbf{Credit}^{\boldsymbol{T}}$ is a vector with 201,619 components, one for each feature.

I reshaped the first 201,600 components of $\mathbf{Credit}^{\boldsymbol{T}}$ to a $105 \times 80 \times 24$ tensor, where the 24 channels are the credit assigned to the binned values of the RGB channels of the game frame. I then summed over the 24 channels to get a 105×80 matrix. I visualize this matrix for each game in Figure 7.9 as an imperfect way of visualizing the credit assigned to each pixel location.

Figure 7.9 shows that SwiftTD assigned credit to meaningful aspects of the game that are predictive of rewards and returns. For example, in Pong, it assigned credit to the 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. The visualization serves as a qualitative validation of the credit assignment mechanism in SwiftTD.

Chapter 8

Swift-Sarsa: Extending SwiftTD to Control

SwiftTD can learn predictions more accurately than prior TD learning algorithms. The ideas that enable it to learn better predictions can be applied to control algorithms as well. The most straightforward way of applying insights from SwiftTD to control problems is to combine its key ideas with True Online Sarsa(λ) (Van Seijen et al., 2016) to develop *Swift-Sarsa*.

8.1 Swift-Sarsa: Fast and Robust Linear Control

In the control problem outlined in Chapter 3, the output of the agent at every time step is a vector with d components. Swift-Sarsa is limited to problems with a discrete number of actions. If each component of the action vector can only have a finite number of values, then we can represent the problem as having a discrete set of actions.

Swift-Sarsa uses SwiftTD to learn a value function for each of its m discrete actions. At every time step, it computes the value of each action and stacks them to get an action-value vector. A policy function $\pi : \mathbb{R}^m \to \{1, \cdots, m\}$ takes as input the action-value vector and returns a discrete action. The value of the action chosen at the current time step is used in the bootstrapped target, and the value of the action chosen at the previous step is used as a prediction when estimating the TD error. The eligibility trace vector of the

value function of only the chosen action is incremented.

We can make the description of the algorithm more concrete with some notation. Let $\boldsymbol{w^i}_t$ be the weight parameter vector for the value function for action i at time step t, and let $\boldsymbol{\phi_t}$ be the feature vector at time step t. The value of the jth action is:

$$v^j_{t-1,t} = \sum_{i=1}^{n} w^j_{t-1}[i]\phi_t[i]. \tag{8.1}$$

The values associated with all actions are stacked to form the action-value vector $\boldsymbol{v_{t-1,t}} \in \mathbb{R}^m$ where

$$v_{t-1,t}[j] = v^j_{t-1,t} \text{ for } j \in \{1, \cdots, m\}. \tag{8.2}$$

Let a_t and a_{t-1} be the actions chosen at time step t and $t-1$, respectively. The TD error in Swift-Sarsa is

$$\delta'_t = r_t + \gamma \boldsymbol{v}_{t-1,t}[a_t] - \boldsymbol{v}_{t-2,t-1}[a_{t-1}]. \tag{8.3}$$

The eligibility vector for the value function of action j is $\boldsymbol{z}^j$. If the action chosen at time step t is j then $\boldsymbol{z}^j$ is decayed by $\lambda\gamma$ and incremented using the same update as True Online TD(λ). If the action chosen is different from j, then the components of $\boldsymbol{z}^j$ are decayed by $\lambda\gamma$ but not incremented. Other than these changes, Swift-Sarsa is the same as SwiftTD. Algorithm 15 is the pseudocode of Swift-Sarsa.

The policy π can be any function. Usually, it is chosen such that actions with high values are more likely to be picked than actions with low values. Two popular choices of policies are the ϵ-greedy policy and the softmax policy.

The ϵ-greedy policy picks the action with the highest value with probability $1-\epsilon$ and a random action with probability ϵ. The softmax policy turns the values into a discrete probability distribution. The probability of taking the ith action is

$$\frac{e^{\frac{v_t[i]}{\tau'}}}{\sum_{j=1}^{m} e^{\frac{v_t[j]}{\tau'}}}, \tag{8.4}$$

where $\tau' \in (0, \infty)$ is the *temperature* parameter. Changing the temperature parameter does not change the relative order of likelihood of actions. For a

fixed action-value vector, a high value of the temperature parameter makes the policy closer to a uniform policy, and a low value of the temperature parameter makes the policy closer to the greedy policy. In the limit when $\tau' \to \infty$, the probability of every action is the same, and when $\tau' \to 0$, the probability of action with the highest value is one.

8.2 The Operant Conditioning Benchmark

I designed a test bed called *the operant conditioning benchmark* for evaluating the performance of Swift-Sarsa. The benchmark defines a set of control problems that do not need sophisticated strategies for exploration, and a random policy picks the best actions occasionally. The optimal policy for problems from the benchmark can be represented by a linear learner.

The inspiration for the operant conditioning benchmark is the *animal learning benchmark* by Rafiee et al. (2023). The animal learning benchmark is inspired by classical conditioning experiments done by behaviorists on animals, and the operant conditioning benchmark is inspired by operant conditioning experiments. The key difference between them is that in operant conditioning experiments, the actions chosen by the animals influence the rates of the rewards. In classical conditioning experiments the animals have no control over the rates of rewards and simply learn to predict the upcoming rewards (*e.g.*, Pavlov's dog).

The observation vectors in problems in the benchmark have n binary components, and the action-vectors have d binary components. Both n and d are hyperparameters, and any combination of them for which $n > d$ defines a valid control problem.

At some special time steps, exactly one of the first m components of the observation vector is one. They are zero on all other time steps. On time steps when the ith component of the first m components is one, the agent gets a delayed reward for picking an action-vector whose ith component is one and other components are zero. The reward is delayed by k_1 steps, where k_1 is a variable that is uniformly sampled from (ISI_1, ISI_2) every time the agent

picks the rewarding action. On all other time steps, the reward is zero.

One randomly chosen component from the first m components of the observation vector are one every k_2 time steps, where k_2 is a variable that is uniformly sampled from (ITI_1, ITI_2)

At every step, each of the remaining $n - m$ components of the observation vector is one with probability μ_t. $\mu_1 = 0.05$, and it is recursively updated as

$$\mu_t = \begin{cases} \mu_{t-1} + & \text{if } 0.01 \leq \mu_{t-1} + n_t \leq 0.1 \\ 0.01 & \text{if } \mu_{t-1} + n_t < 0.01 \\ 0.1 & \text{if } \mu_{t-1} + n_t > 0.1, \end{cases} \tag{8.5}$$

where $n_t \sim \mathcal{N}(0, 10^{-8})$.

Intuitively μ is the value of a random walk that starts at 0.05, and it is updated by adding a sample from $\mathcal{N}(0, 10^{-8})$ at every time step. μ is forced to stay in the range $[0.01, 0.1]$.

The last $n - m$ components of the observation vector are a source of noise with a time-dependent distribution. Control problems whose observations have many noisy components ($n - m$ is large) are challenging.

8.3 Experiments: Swift-Sarsa on the Operant Conditioning Benchmark

I ran experiments with Swift-Sarsa on the operant conditioning benchmark for different values of n. I set $m = 2$ in all experiments. (ISI_1, ISI_2) was $(4, 6)$, and (ITI_1, ITI_2) was (50, 70). The lifetime of the agent was 300,000. The policy was softmax with a temperature parameter of 0.1. The action-vector is mapped to a discrete set. Actions (0, 0), (0, 1), (1, 0), and (1, 1) are mapped to discrete actions 1, 2, 3, and 4, respectively.

Figure 8.1 plots the average reward for different values of the meta-step-size parameter and the initial value of the step-size parameters for two different values of n. Similar to the performance of SwiftTD, the performance of Swift-Sarsa improved as the meta-step-size parameter increased showing the benefit of step-size optimization. For a wide range of its parameters, Swift-Sarsa achieved a lifetime reward that was close to the optimal lifetime reward of

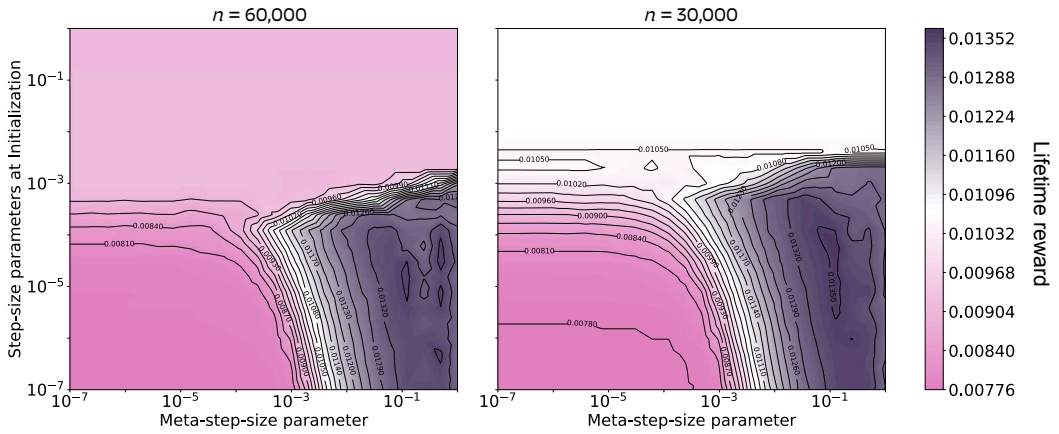

Figure 8.1: Performance of Swift-Sarsa as a function of the meta-step-size parameter and the initial values of step-size parameters on the operant conditioning benchmark. Experiments in the left figure had $n = 60,000$ and the right figure had $n = 30,000$. For both set of experiments η was 1.0, m was 2, and ϵ was 0.9999.

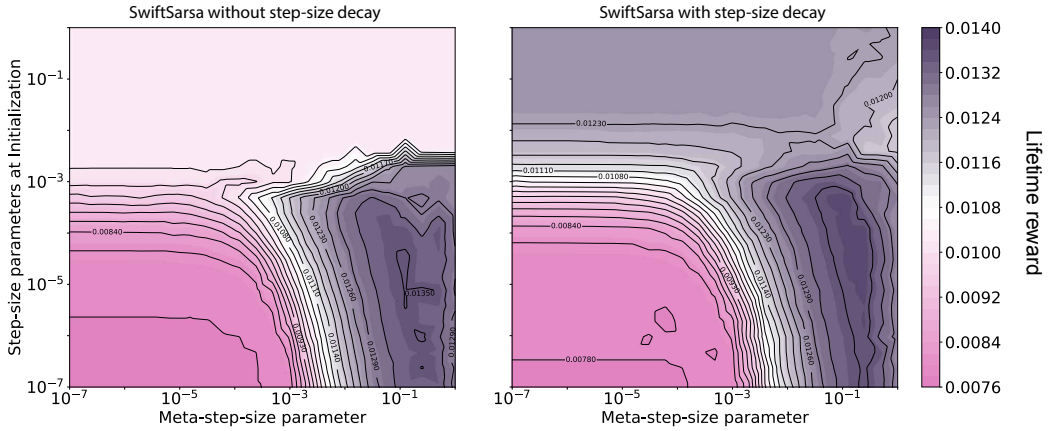

Figure 8.2: Impact of step-size decay on the performance of Swift-Sarsa as a function of the meta-step-size parameter and the initial values of step-size parameters on the operant conditioning benchmark. Experiments in the left panel did not use step-size decay whereas experiments in the right panel used a step-size decay with decay parameter set to 0.999. Comparing the two results we see that step-size decay improves performance when the initial value of the step-size parameters is too large. For both sets of experiments, η was one and m was two.

≈ 0.014. Increasing the number of distractors made the problem more challenging, and the performance of Swift-Sarsa decreased.

In a second set of experiments, I compared the impact of step-size decay on the performance of Swift-Sarsa. The results are in Figure 8.2. Similar to its impact on SwiftTD, step-size decay improved the performance of Swift-Sarsa when the initial value of the step-size parameters was too large.

Swift-Sarsa is a simple way of transferring the improvements made by SwiftTD to control problems. A more thorough evaluation of Swift-Sarsa on a wider range of control problems is needed to understand its full potential. It is possible that Swift-Sarsa when combined with more powerful preprocessing, such as *tile coding* (Sutton & Barto, 2018), can perform similarly to deep RL algorithms on more complex problems, such as Atari games.

Algorithm 15: Swift-Sarsa

Hyperparameters: $\epsilon = 0.999, \eta = 0.1, \eta^{min} = e^{-15}, \alpha^{init} = 10^{-7}, \gamma, \lambda, \theta$
Initializations: $\boldsymbol{w}, \boldsymbol{h^{old}}, \boldsymbol{h^{temp}}, \boldsymbol{z^{\delta}}, \boldsymbol{p}, \boldsymbol{h}, \boldsymbol{z}, \bar{\boldsymbol{z}} \leftarrow \mathbf{0} \in \mathbb{R}^n; (v^{\delta}, v^{old}) = (0,0); \boldsymbol{\beta} \leftarrow \ln(\boldsymbol{\alpha}^{init}) \in \mathbb{R}^n$
while *alive* **do**
  Perceive $\boldsymbol{\phi}$ and r
  for $i \in 0, \cdots, m$ **do**
    $v[i] \leftarrow \sum_{j|\phi[j]\neq 0} w^i[j]\phi[j]$
  $k \leftarrow \pi(\boldsymbol{v})$
  $\delta' \leftarrow r + \gamma v[k] - v^{old}$
  for $i \mid z^j[i] \neq 0 \ \forall i, j$ **do**
    $\delta^{w^j}[i] \leftarrow \delta' z^j[i] - z^{\delta^j}[i] v^{\delta}$
    $w^j[i] \leftarrow w^j[i] + \delta^{w^j}[i]$
    $\beta^j[i] \leftarrow \beta^j[i] + \frac{\theta}{e^{\beta^j[i]}}(\delta' - v^{\delta}) p^j[i]$
    $\beta^j[i] \leftarrow \text{clip}\left(\beta^j[i], \ln(\eta^{min}), \ln(\eta)\right)$
    $h^{old^j}[i] \leftarrow h^j[i]$
    $h^j[i] \leftarrow h^{temp^j}[i] + \delta' \bar{z}^j[i] - z^{\delta^j}[i] v^{\delta}$
    $z^{\delta^j}[i] = 0$
    $(z^j[i], p^j[i], \bar{z}^j[i]) \leftarrow (\gamma\lambda z^j[i], \gamma\lambda p^j[i], \gamma\lambda \bar{z}^j[i])$
  $v^{\delta} \leftarrow 0$
  $\tau \leftarrow \sum_{i|\phi[i]\neq 0} e^{\beta^k[i]} \phi[i]^2$
  $b \leftarrow \sum_{i|\phi[i]\neq 0} z^k[i]\phi[i]$
  for $i \mid \phi[i] \neq 0$ **do**
    $v^{\delta} \leftarrow v^{\delta} + \delta^{w^k}[i]\phi[i]$
    $z^{\delta^k}[i] \leftarrow \min\left(1, \frac{\eta}{\tau}\right) e^{\beta^k[i]}\phi[i]$ `// η-bound`
    $z^k[i] \leftarrow z^k[i] + z^{\delta^k}[i](1-b)$
    $p^k[i] \leftarrow p^k[i] + \phi[i] h^{old^k}[i]$
    $\bar{z}^k[i] \leftarrow \bar{z}^k[i] + z^{\delta^k}[i]\left(1 - b - \phi[i]\bar{z}^k[i]\right)$
    $h^{temp^k}[i] \leftarrow h^k[i] - z^{\delta^k}[i]\phi[i]h^k[i] - h^{old^k}[i]\phi[i](z^k[i] - z^{\delta^k}[i])$
    if $\tau > \eta$ **then**
      $\beta^k[i] = \beta^k[i] + \phi[i]^2 \ln(\epsilon)$ `// Step-size decay`
      $(h^{temp^k}[i], h^k[i], \bar{z}^k[i]) = (0,0,0)$
  $v^{old} \leftarrow v[k]$

Part II

Fast Non-linear Recurrent Feature Discovery

Chapter 9

Feature Generation by Continual Imprinting

In Part 1 of the dissertation I presented algorithms that can learn from a given set of features. I did not answer the question of how to get the features. In this chapter, I propose algorithms for finding useful features from observations.

The chapter is organized as follows: first I outline a general architecture of a learning system that continually generates new features, continually removes useless features, and continually learns predictions from the features. I then explain the idea of feature generation by *imprinting* and present two algorithms for generating features. Finally, I conclude by evaluating the proposed learning system on a new prediction benchmark that uses real-world audio data as observations.

9.1 Learning by Feature Generation and Feature Removal

The learning system that continually generates and removes features is called the *imprinting learner*, and it has three parts: a feature generator, a prediction learner, and a feature remover. Let $\boldsymbol{x_t}$ and $\boldsymbol{\phi_t}$ be the observation vector and state-feature vector at time step t. Then $\boldsymbol{\phi_t}$ is computed as:

$$\boldsymbol{\phi_t} = U(\boldsymbol{\phi_{t-1}}, \boldsymbol{x_t}), \tag{9.1}$$

where $\boldsymbol{x_t} \in \{0, 1\}^n$ is a binary-valued observation vector and U is the *state update function*. Unlike agents in Part I of the dissertation that learn from

feature vectors with a constant length, the agents in Part II have feature vectors that can grow and shrink over time. At time step t the feature vector has n_t components.

At time step t, the imprinting learner computes its feature vector $\boldsymbol{\phi}_t$ and uses it to make a prediction:

$$v_{t-1,t} = \sum_{i=1}^{n_t} w_{t-1}[i]\phi_t[i], \tag{9.2}$$

where $\boldsymbol{w_{t-1}}$ is the weight parameter vector. Every feature has an associated weight parameter, a step-size parameter, and an eligibility trace parameter that are all updated by SwiftTD.

Features have one of the three status, *tenure-track*, *tenured*, and *idle*. They also have two hyperparameters associated with them, *tenure-threshold* and *tenure-track-threshold*. Any feature whose magnitude of the weight parameter (*i.e.*, the parameter used to make predictions) is larger than or equal to the tenure-threshold parameter has the tenured status. Any feature whose weight parameter is larger than or equal to the tenure-track-threshold parameter and less than the tenure-threshold parameter has the tenure-track status. Lastly, any feature whose weight parameter is less than the tenure-track-threshold parameter has the idle status. The status of a feature can change over time as its weight parameter changes. The values of the threshold parameters can be different for different features.

In addition to having a status, each feature has a type. It can be an *observation feature*, a *pattern feature*, or a *memory feature*. Observation features are present at initialization and never removed. Pattern features and memory features are generated from experience and can be removed. All types of features are binary-valued (0 or 1).

At initialization, the agent has n observation features. These features have the same values as the observation vector $\boldsymbol{x}$ at every step. Their weight parameters are initialized to zero and their step-size parameters are initialized to α^{init}, where α^{init} is a hyperparameter of SwiftTD.

Feature Generation

The imprinting learner generates up to k new features at every time step where k is a hyperparameter. Recall that SwiftTD has a parameter η that limits the increment to its eligibility trace vector. New features are generated at time step t as long as the value of τ_t is less than η. The value of τ_t for a learner with binary features is:

$$\tau_t = \sum_{i|\phi_t[i]=1} e^{\beta}[i]. \tag{9.3}$$

If τ_t is less than $\eta + \alpha^{init}$ and $\boldsymbol{\phi_{t-1}}$ has active tenured features, then a new feature is generated. If the new feature is not identical to any of the existing features, then it is added to the learner. Its weight parameter and the step-size parameter are initialized to 0 and α^{init}, respectively. Lastly, if the new feature has a value of one at the current time step, then τ_t is updated as:

$$\tau_t = \tau_t + \alpha^{init}. \tag{9.4}$$

If at any point during feature generation, no more unique feature can be generated, k features have been generated, or $\tau_t + \alpha^{init}$ is greater than η, then feature generation stops.

Feature generation adds up to k features but only if the feature vector on the previous time step had active tenured features, and τ after all the features have been added does not exceed η.

Feature Removal

SwiftTD (Algorithm 14) decays the ith component of the eligibility trace parameter as:

$$z[i] \leftarrow \gamma\lambda z[i]. \tag{9.5}$$

After the decay, if the value of $z[i]$ is less than $e^{\beta}[i]\epsilon^z$ and the status of the feature $\phi[i]$ is idle, then the imprinting learner removes $\phi[i]$ and replaces it by the last component of the feature vector. Here ϵ^z is a hyperparameter which can be set to a small value, such as 0.01. Once the feature is removed the length of the feature vector is reduced by one. The observation features are

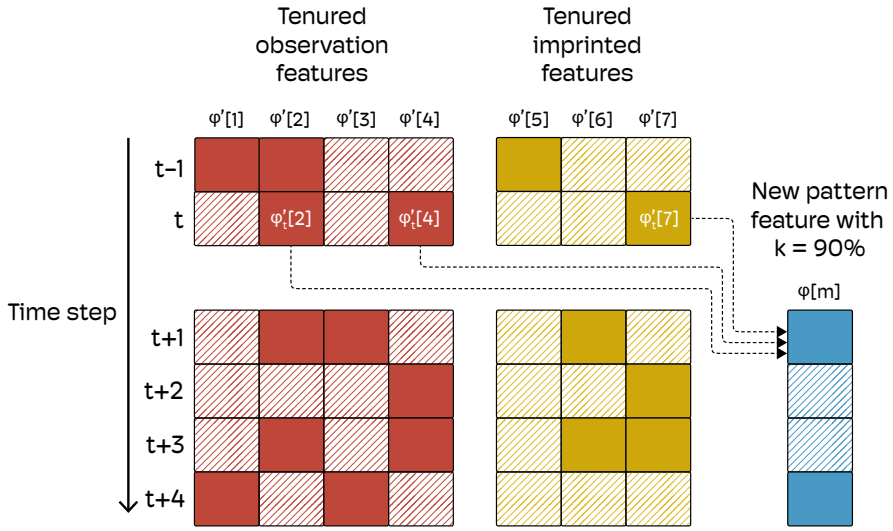

Figure 9.1: Generating a new feature at time step $t+1$ by imprinting on the values of the tenured features at time step t. The solid-colored features are one (active) and the striped ones are zero (not active). Here $\boldsymbol{\phi}'_t$ is the vector of tenured features at time step t. The new feature is connected to $\phi'[2], \phi'[4]$ and $\phi'[7]$ and is active at time step j if $\phi'_{j-1}[2] + \phi'_{j-1}[4] + \phi'_{j-1}[7]$ is greater than or equal to 2.7 (90% of 3.0), which only happens when all three of the input features are active.

never removed because their tenure-track threshold is zero and they never have the idle status.

A subtle but important point is that features removed from the feature vector can have outgoing connections to other features in the feature vector, and they may still be required to update the feature vector. These *ghost features* do not have direct connections to predictions but are still part of the state update function. The state update function can become more complex as the number of ghost features increases without increasing the size of the feature vector.

9.2 Feature Generation by Imprinting

The idea of imprinting is to generate features at time step t that are immediately triggered by their inputs. Making features that are immediately triggered has two benefits. First, the usefulness of these features can be tested immediately, and second, restricting the learner to only generate these features culls

the search space of the features. In this chapter, I look at two mechanisms of generating features by imprinting. The first mechanism generates features that recognize the pattern of the feature vector that they imprint on. They are called *pattern features*. The second mechanism generates features that remember the past. These are called *memory features*.

Generating Pattern Features by Imprinting

Pattern features are generated to recognize the current configurations of the tenured features. The idea is visually shown in Figure 9.1. A new pattern feature is constructed by connecting it to non-zero tenured features such that the new feature is one when at least k_0 percent of the same tenured features are one, where k_0 is a hyperparameter of the generator. A pattern feature is always active the first time step it is created.

Generating Memory Features by Imprinting

Memory features are generated to remember and relay information from the past to the future. The idea is visually shown in Figure 9.2. A new memory feature is constructed by connecting it to a single tenured feature. When the tenured feature is one, the memory feature is one for k_1 time steps after a delay of k_2 time steps, where k_1 and k_2 are hyperparameters of the generator. It is zero otherwise. A memory feature is always triggered the first time step it is created but is not active until the delay has passed.

9.3 The Audio Prediction Benchmark

The audio prediction benchmark uses audio recorded from a microphone as the data stream. It consists of sounds, such as words, that are followed by scalar rewards after a short delay. An example of the data stream with labels is in Figure 9.3. Every experiment has three sounds. One is followed by a +1 reward, one is followed by a -1 reward and one is not followed by any rewards. The sounds differ for different instantiations of prediction problems. Some examples of the sounds are spoken words and notes produced from different

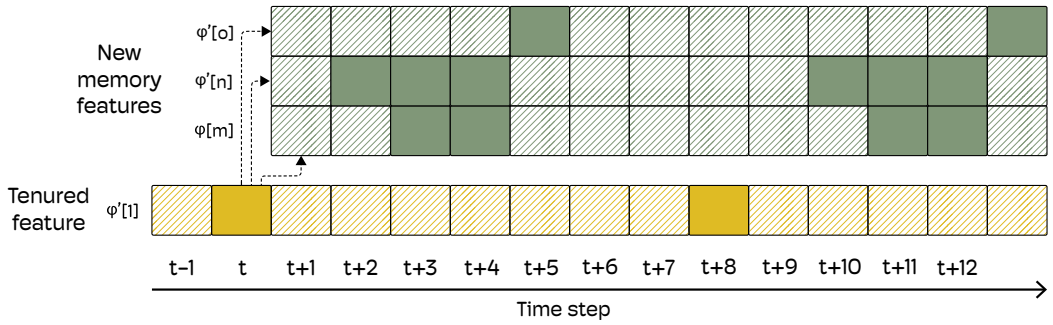

Figure 9.2: Generating three memory, $\phi[m], \phi[n]$, and $\phi[o]$, from the tenured feature $\phi'[1]$. The three features are triggered by $\phi'_t[1]$ and $\phi'_{t+8}[1]$. When triggered, $\phi[m]$ is active for two time steps with a delay of two time steps, $\phi[n]$ is active for three time steps with a delay of one time step, and $\phi[o]$ is active for one time steps with a delay of three time steps.

musical instruments.

Learning accurate predictions on the audio prediction benchmark requires non-linear features that differentiate between sounds. Some sounds are deliberately chosen to be similar, such as the same chord played on two different instruments. The delay between the sound and the reward signal is wider than the duration of the sound. This means that to make accurate predictions an agent must remember information from its past observations. The benchmark is designed to test the ability of a learner to generate features that can recognize patterns and retain information from the past.

Preprocessing to get binary observations

The audio for the benchmark was sampled at a rate of 16,384 Hz for one hour. At every environment step it progresses by 640 samples or roughly 40 milliseconds. The samples are transformed into the frequency domain using *the fast fourier transform* (FFT) algorithm, which is applied to the last 1024 samples (640 from the current time step and 384 from the previous time step; there is a small overlap between every two time steps because most FFT implementations like to work in powers of two).

The output of FFT is a vector with 512 components. The ith component is the magnitude of the signal of frequency i. This vector is further processed to get a binary observation vector.

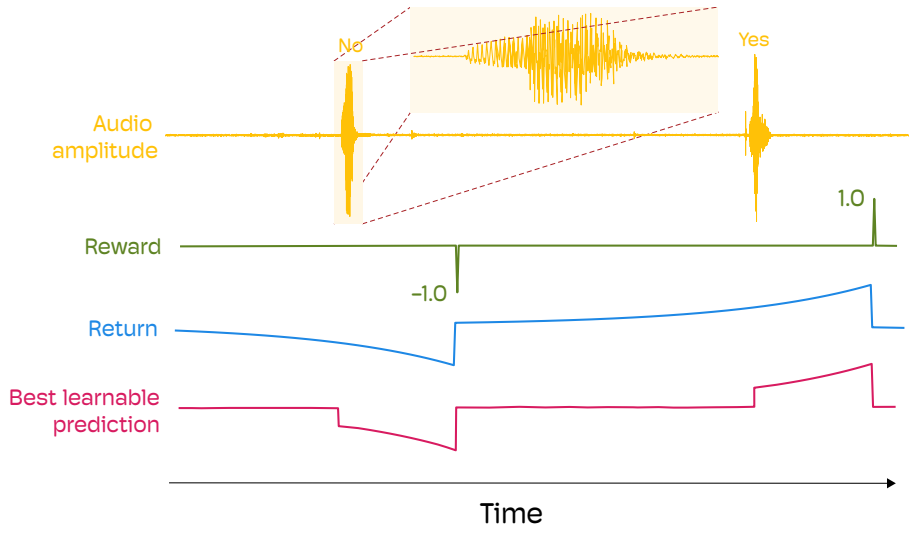

Figure 9.3: Visualizing experience from the audio prediction benchmark. The sound of the word *no* is followed by a reward of -1 after a delay of 3 to 5 seconds, and the sound of the word *yes* is followed by a reward of +1 after a similar delay. The delay between the sounds of the two words is 15 to 30 seconds. The return cannot be perfectly predicted from the audio signal, and the best learnable prediction starts after the sound is audible.

The output of FFT is plotted on a graph with frequency on the x-axis and magnitudes of the signal on the y-axis. The range of the x-axis is 1 to 512 and the range of the y-axis is 0 to 50. The plot is divided into a grid of 50 $\times$ 50 equal squares. Each square is a component of the observation vector. It is one if the frequency plot passes through it and zero otherwise. The output of the preprocessing is a binary observation vector of size 2500 with exactly 50 components that are one.

Prediction problems in the benchmark

I use three prediction problems in experiments. The problems are identical to each other in all aspects except the sounds. One uses the sounds of the word yes (followed by +1 reward), no (followed by -1 reward), and maybe (not followed by a reward), and the second uses the sound of C chords strummed on a guitar (followed by +1 reward), the sound of C chord played on a piano (followed by -1 reward), and the sound of a D chord played on a piano (not followed by a reward). The third uses the sound of the note C4 played on

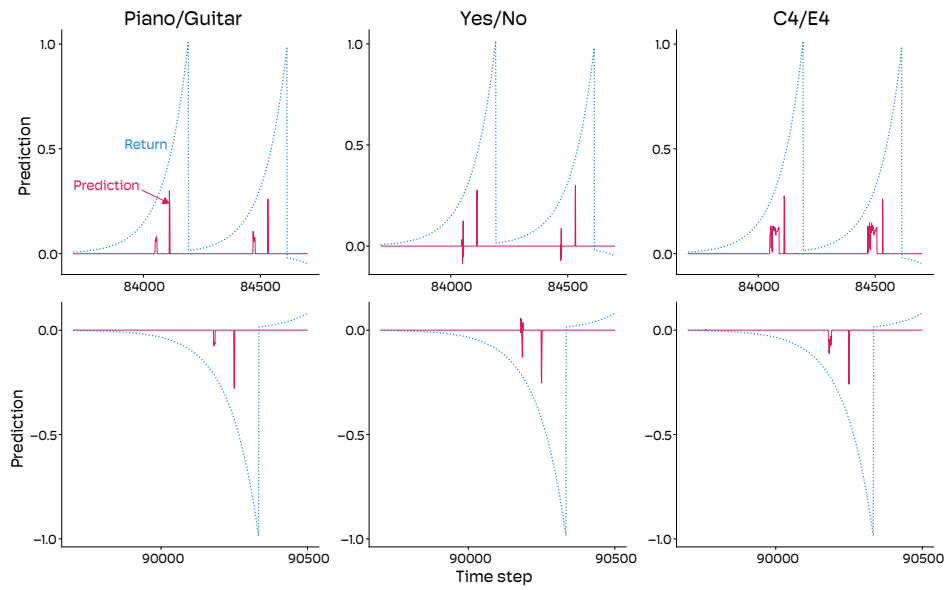

Figure 9.4: Predictions learning by SwiftTD on the three problems from the Audio Prediction Benchmark. SwiftTD only predicted the return momentarily, likely when the sound was still audible.

a piano (followed by +1 reward), note E4 played on a piano (followed by -1 reward), and note D4 played on a piano (not followed by a reward). The same sound signal is never used twice that is, all sounds are unique. I call the three problems *Yes/No*, *Guitar/Piano*, and *C4/E4* .

9.4 Experiment: SwiftTD on the Audio Prediction Benchmark

As a sanity check, I first ran SwiftTD on all three problems. The predictions learned at the end of learning by the best-performing learner are in Figure 9.4. The top row has predictions for positive rewards and the bottom row has predictions for negative rewards.

SwiftTD did not learn to accurately predict the rewards. In all three problems, it predicted the return well for a fraction of a second, perhaps at times when the sound was still audible. The predictions dropped to zero quickly. This is expected because SwiftTD is predicting directly from observations and has no way to remember the sounds after they are not audible.

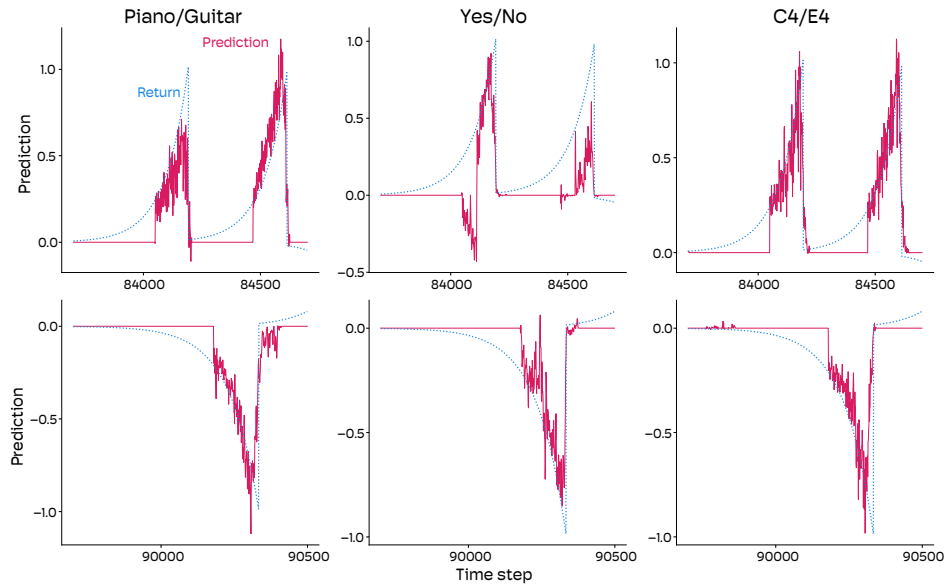

Figure 9.5: Predictions learned by imprinting learner on the three problems from the Audio Prediction Benchmark. In all three problems, it learned to predict the onset of rewards, and the predictions are sustained until the reward.

9.5 Experiments: Imprinting Learner on the Audio Prediction Benchmark

I ran the imprinting learner on the audio prediction benchmark. The agent generated up to ten memory features and ten pattern features at every time step. The initial step-size parameter was 3^{-3}, the tenue-track-threshold parameter was 3^{-4}, and the tenure-threshold parameter was 0.01. The ϵ^z was 0.01. The agent learned for 96,000 steps, which was roughly 1 hour of audio data.

The hyperparameters of the memory feature generator, k_1, and k_2, were sampled from the range 1 to 3 and 0 to 20, respectively, for every generation, and the hyperparameter for the pattern feature generator, k_0, was sampled from the set $\{60, 70, 80, 90\}$ for every generation.

I plot the predictions learned by the agent at the end of learning in Figure 9.5. The agent learned to predict the returns better than the SwiftTD learner. The predictions were more accurate on the Piano/Guitar and C4/E4 problems than the predictions on the Yes/No problem.

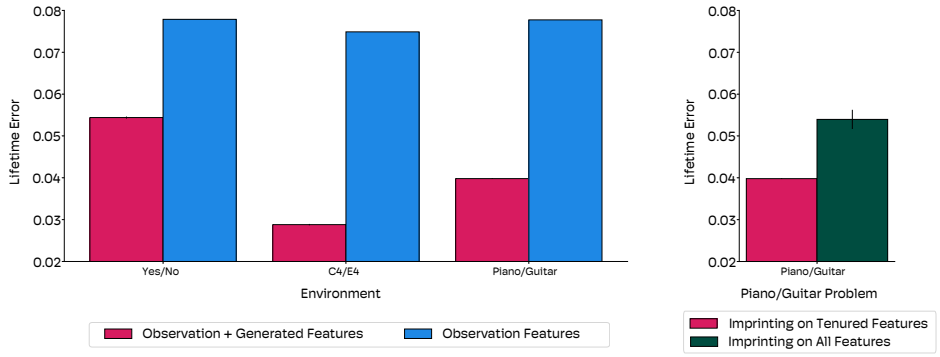

Figure 9.6: Performance of imprinting learner on the audio prediction benchmark. The bar plots show the mean lifetime error over fifty seeds. The error bars are +- standard error. In the left panel the imprinting learner is compared to SwiftTD. In the right panel two versions of the imprinting learner are compared. One imprints on active tenured features, and the other imprints on all active features.

I repeated the above experiment with fifty different seeds. The results of the average lifetime error over all seeds are in the left panel of Figure 9.6. The plots have error bars of $\pm$ standard errors. Feature generation by imprinting resulted in a lower lifetime error on all three problems than a learner that did not generate features. I repeated the experiments with a version of the imprinting learner that imprints on all active features and not just the tenured active features. The results are in the right panel of Figure 9.6. The imprinting learner that imprinted on all active features performed worse than the one that imprinted on just the tenured features.

I also looked at the total number of features generated by the learners, and the total number of features kept by the learners at the end of their lifetime. In the Piano/Guitar problem, the best-learner generated 26,288 features throughout its lifetime. At the end of learning, it had 2007 tenured features and 4265 tenure-track features. Of the tenured features, 1828 were memory features, 115 were pattern features, and 64 were observation features. The numbers for other problems were qualitatively similar. The imprinting learner that imprinted on all active features generated 398,328 features and had 7093 tenured and 334,649 tenure-track features at the end.

These numbers are interesting because the imprinting learners were able

to search over tens of thousands of features and find an order of magnitude smaller subset of them that were useful for the prediction task, demonstrating their effectiveness. The learners that imprinted on all active features were less efficient in their search and generated an order of magnitudes more features and still performed worse.

Feature generation by imprinting is a promising solution for quickly generating features that can recognize patterns and remember the past. It is unlikely that the one-shot feature generation by imprinting would be capable of learning nuanced patterns. A more complete algorithm would continually generate new features and adapt existing features. Gradient-based learning provides a promising way to adapt features. In the next chapter, I present algorithms that can adapt recurrent features online in a computationally efficient manner.

Chapter 10

Feature Tuning using Columnar-Constructive Networks

Algorithms discussed so far do not adapt features over the lifetime of the agent. In Chapters 5, 6, 7, and 8, the feature vector was given to the agent, and the agent could not change it. In Chapter 9, the agent could construct features over time but not adapt them. Once a feature had been constructed it could only be removed. In most real situations, it is naive to expect that we can handcraft the features or construct them in one shot. A more powerful learning system would be one that continually improves existing features using experience. In this chapter, I present recurrent learning algorithms that can adapt features over time in a computationally efficient manner.

Let the agent be a recurrent network that has n features, $\boldsymbol{\phi} \in \mathbb{R}^n$. At every time step, it combines the feature vector with a weight vector to make a scalar prediction. The feature vector is computed by a *state update function* U parameterized by a learnable parameter vector $\boldsymbol{\theta}$. The features at time step t are computed as:

$$\boldsymbol{\phi}_t = U(\boldsymbol{\phi}_{t-1}, \boldsymbol{x}_t, \boldsymbol{\theta}), \tag{10.1}$$

where $\boldsymbol{x}_t$ is the observation vector, and $\boldsymbol{\phi_{t-1}}$ is the feature vector at time step $t-1$.

The features are linearly combined with a weight parameter vector $\boldsymbol{w}_{t-1} \in$

$\mathbb{R}^n$ to make a prediction $v_{t-1,t}$ as:

$$v_{t-1,t} = \sum_{k=1}^{n} w_{t-1}[i]\phi_t[i] \tag{10.2}$$

A sensible choice of U is a differentiable recurrent neural network. Its parameters can be updated using gradient-descent as:

$$\boldsymbol{\theta'} = \boldsymbol{\theta} - \alpha \frac{\partial (v_{t-1,t} - y_t^*)^2}{\partial \boldsymbol{\theta}}, \tag{10.3}$$

where y_t^* is the target, and α is the step-size parameter. The gradient can be expanded as:

$$\frac{\partial (v_{t-1,t} - y_t^*)^2}{\partial \boldsymbol{\theta}} = \frac{\partial (v_{t-1,t} - y_t^*)^2}{\partial v_{t-1,1}} \frac{\partial v_{t-1,t}}{\partial \boldsymbol{\phi}_t} \frac{\partial \boldsymbol{\phi}_t}{\partial \boldsymbol{\theta}}. \tag{10.4}$$

The key question is how to compute $\frac{\partial \boldsymbol{\phi}_t}{\partial \boldsymbol{\theta}}$. We can obtain a recursive formula for this expression, which is used by RTRL (Williams & Zipser, 1989) and by the algorithms explained in this chapter. RTRL assumes that the parameters of the recurrent network are kept fixed over time. We make the same assumption, and as a result, $\boldsymbol{\theta}$ is not indexed by time. When using the algorithms in experiments, we break our assumption and update the parameters of the network at every time step using Equation 10.3.

To make it clear how we can use the multivariable chain rule, let us rewrite the state update function as $\boldsymbol{\phi}_t = U\left(\boldsymbol{\phi}_{t-1}(\boldsymbol{\theta}), \boldsymbol{x}_t, \boldsymbol{g}_t(\boldsymbol{\theta})\right)$ where $\mathbf{g}_t(\boldsymbol{\theta}) \doteq \boldsymbol{\theta}$. Then the multivariable chain rule gives us:

$$\frac{\partial \boldsymbol{\phi}_t}{\partial \boldsymbol{\theta}} = \frac{\partial \boldsymbol{\phi}_t}{\partial \mathbf{g}_t} \frac{\partial \mathbf{g}_t}{\partial \boldsymbol{\theta}} + \frac{\partial \boldsymbol{\phi}_t}{\partial \boldsymbol{\phi}_{t-1}} \frac{\partial \boldsymbol{\phi}_{t-1}}{\partial \boldsymbol{\theta}}, \tag{10.5}$$

where the first term in the sum is the gradient of the state of the network under the assumption that $\boldsymbol{\phi}_{t-1}$ is not a function of $\boldsymbol{\theta}$, and the second term takes into account the indirect impact of $\boldsymbol{\theta}$ on $\boldsymbol{\phi}_t$ due to its impact on $\boldsymbol{\phi}_{t-1}$.

This recursive relationship is exploited by two algorithms: BPTT and RTRL. BPTT stores all past feature vectors and observation vectors and ex-

pands equation 10.3 as:

$$\begin{aligned}
\frac{\partial v_{t-1,t}}{\partial \boldsymbol{\theta}} &= \frac{\partial v_{t-1,t}}{\partial \boldsymbol{\phi}_t}\frac{\partial \boldsymbol{\phi}_t}{\partial \boldsymbol{\theta}} \\
\frac{\partial v_{t-1,t}}{\partial \boldsymbol{\theta}} &= \frac{\partial v_{t-1,t}}{\partial \boldsymbol{\phi}_t}\frac{\partial \boldsymbol{\phi}_t}{\partial \mathbf{g}_t}\frac{\partial \mathbf{g}_t}{\partial \boldsymbol{\theta}} + \frac{\partial v_{t-1,t}}{\partial \boldsymbol{\phi}_t}\frac{\partial \boldsymbol{\phi}_t}{\partial \boldsymbol{\phi}_{t-1}}\frac{\partial \boldsymbol{\phi}_{t-1}}{\partial \boldsymbol{\theta}} \\
&= \frac{\partial v_{t-1,t}}{\partial \boldsymbol{\phi}_t}\frac{\partial \boldsymbol{\phi}_t}{\partial \mathbf{g}_t}\frac{\partial \mathbf{g}_t}{\partial \boldsymbol{\theta}} + \frac{\partial v_{t-1,t}}{\partial \boldsymbol{\phi}_t}\frac{\partial \boldsymbol{\phi}_t}{\partial \boldsymbol{\phi}_{t-1}}\frac{\partial \boldsymbol{\phi}_{t-1}}{\partial \boldsymbol{g}_{t-1}}\frac{\partial \boldsymbol{g}_{t-1}}{\partial \boldsymbol{\theta}} + \frac{\partial v_{t-1,t}}{\partial \boldsymbol{\phi}_t}\frac{\partial \boldsymbol{\phi}_t}{\partial \boldsymbol{\phi}_{t-1}}\frac{\partial \boldsymbol{\phi}_{t-1}}{\partial \boldsymbol{\phi}_{t-2}}\frac{\partial \boldsymbol{\phi}_{t-2}}{\partial \boldsymbol{\theta}},
\end{aligned} \tag{10.6}$$

to compute the gradient. It unrolls the recursive expansion back in time, computing and accumulating gradient until the start of the recursion at $t = 0$. RTRL, on the other hand, computes the Jacobian $\frac{\partial \boldsymbol{\phi}_t}{\partial \boldsymbol{\theta}}$ incrementally by using Equation 10.5 at every time step. To get the gradient w.r.t the prediction, it uses Equation 10.4. Both algorithms compute the same gradient but make different compromises in terms of computation and memory.

RTRL does not store past feature vectors and observation vectors as it can update the Jacobian using only information from the current time step. However, computing the Jacobian using Equation 10.5 requires $O(|\boldsymbol{\phi}|^2|\boldsymbol{\theta}|)$ operations and $O(|\boldsymbol{\phi}||\boldsymbol{\theta}|)$ memory. The size of the parameters $|\boldsymbol{\theta}|$ in a fully connected RNN is $|\boldsymbol{\phi}|^2$. RTRL is therefore often said to have quartic complexity in the size of the feature vector.

BPTT requires $O(|\boldsymbol{\theta}|t)$ memory and compute, where t is the length of the sequence. It avoids the bigger memory cost by computing the product $\frac{\partial v_{t-1,t}}{\partial \boldsymbol{\phi}_t}\frac{\partial \boldsymbol{\phi}_t}{\partial \mathbf{g}_t}\frac{\partial \mathbf{g}_t}{\partial \boldsymbol{\theta}}$ directly, rather than separately computing the Jacobian and taking a dot product with $\frac{\partial v_{t-1,t}}{\partial \boldsymbol{\phi}_t}$. For sequences shorter than $|\boldsymbol{\phi}|^2$, BPTT is cheaper than RTRL for fully connected RNNs.

We develop a new approach for recurrent learning called *columnar-constructive networks* (CCNs). CCNs leverage two key ideas: First, RTRL is computationally efficient for modular recurrent networks where each module outputs a single feature; we call these networks *columnar networks*. Second, RTRL is computationally efficient if the recurrent features are learned in stages, as opposed to simultaneously. We call the incremental learning approach *constructive networks*. Figure 10.1 visualizes the central ideas behind columnar networks and constructive networks.

Both columnar networks and constructive networks show promising results but have limitations. Columnar networks cannot learn hierarchical features, and constructive networks cannot learn multiple features in parallel. We show that their weaknesses can be overcome by combining the two ideas to create columnar-constructive networks.

10.1 Columnar Networks

Columnar networks organize the recurrent network such that each scalar recurrent feature is independent of other recurrent features. Let $\phi_t[k]$ be the kth component of the feature vector $\boldsymbol{\phi}_t$. Then, in columnar networks,

$$\phi_t[k] = f_k(\phi_{t-1}[k], \boldsymbol{x}_t, \boldsymbol{\theta}^k). \tag{10.7}$$

The function f_k updates a recurrent feature and is called a column.[1] $\boldsymbol{\theta}^k$ is the parameter vector of the kth column. For any $i \neq j$, the parameter vector $\boldsymbol{\theta}^i$ and $\boldsymbol{\theta}^j$ do not share any components. The outputs of all columns at time step t are concatenated to get the n-dimensional feature vector $\boldsymbol{\phi}_t$. Figure 10.1 (left) shows a graphical representation of a columnar network. Note that changing $\phi[1]$ has no influence on the value of $\phi[2]$ or $\phi[3]$.

Because recurrent features in a columnar network are independent of each other, we can apply RTRL to each of them individually. To better understand why, let us rederive our recursive formula for the gradient. For $\boldsymbol{\theta}^k$, the parameters for the kth column, we have

$$\frac{\partial v_{t-1,t}}{\partial \boldsymbol{\theta}^k} = \frac{\partial v_{t-1,t}}{\partial \boldsymbol{\phi}_t} \frac{\partial \boldsymbol{\phi}_t}{\partial \boldsymbol{\theta}^k} = \sum_{j=1}^{d} \frac{\partial v_{t-1,t}}{\partial \boldsymbol{\phi}_t[j]} \frac{\partial \boldsymbol{\phi}_t[j]}{\partial \boldsymbol{\theta}^k} = \frac{\partial v_{t-1,t}}{\partial \boldsymbol{\phi}_t[k]} \frac{\partial \boldsymbol{\phi}_t[k]}{\partial \boldsymbol{\theta}^k}.$$

All except one term in the summation above are zero because $\boldsymbol{\theta}^k$ does not influence $\boldsymbol{\phi}_t[j]$ when $j \neq k$. Therefore, we only have to compute $\frac{\partial \boldsymbol{\phi}_t[k]}{\partial \boldsymbol{\theta}^k}$ with RTRL. Like before, we can write this recursively using $\phi_t[k] = f\left(\phi_{t-1}[k], \boldsymbol{x}_t, \mathbf{g}_t(\boldsymbol{\theta}^k)\right)$ where $\mathbf{g}_t(\boldsymbol{\theta}^k) \doteq \boldsymbol{\theta}^k$, giving

$$\frac{\partial \phi_t[k]}{\partial \boldsymbol{\theta}^k} = \frac{\partial \phi_t[k]}{\partial \mathbf{g}_t} \frac{\partial \mathbf{g}_t}{\partial \boldsymbol{\theta}^k} + \frac{\partial \phi_t[k]}{\partial \phi_{t-1}[k]} \frac{\partial \phi_{t-1}[k]}{\partial \boldsymbol{\theta}^k}. \tag{10.8}$$

[1] This terminology comes from the connection to structure observed in brains (Mountcastle, 1957).

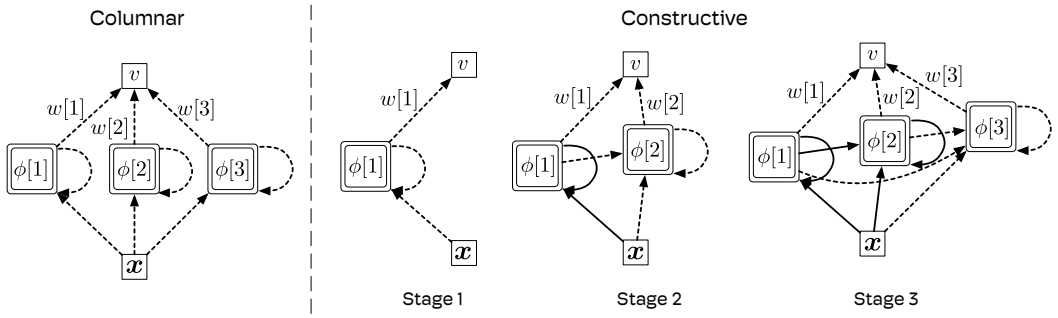

Figure 10.1: Two families of recurrent networks for which gradients can be efficiently computed without bias or noise. Recurrent networks with a columnar structure use $O(n)$ operations and memory per step for learning. However, they do not have hierarchical recurrent features—recurrent features composed of other recurrent features. Constructive networks introduce hierarchical recurrent features and learn them in stages to keep learning computationally efficient.

Computing and storing this Jacobian costs $O(|\boldsymbol{\theta}^k|)$ memory and compute for each column because $|\phi_t[i]| = 1$ for a single column. The cost for all the columns is

$$O(|\boldsymbol{\theta}^1|) + O(|\boldsymbol{\theta}^2|) + \cdots + O(|\boldsymbol{\theta}^n|) = O(|\boldsymbol{\theta}|). \tag{10.9}$$

Therefore, RTRL for columnar networks scales linearly in the size of the parameters.

10.2 Constructive Networks

In constructive networks, we learn the recurrent network one feature at a time. Features learned later can take as input all features learned before them; the opposite is not allowed— features learned earlier cannot take as input features that would be learned later. We elucidate the multi-stage learning process in a small constructive network in Figure 10.1 (right). Dotted lines are parameters that are being updated at every time step, whereas solid lines are parameters that are fixed.

In the first stage, the learner learns the incoming weights of $\phi[1]$ (*i.e.*, $\boldsymbol{\theta}^{\mathbf{1}}$), which is connected to the observation vector $\boldsymbol{x}$, but not to $\phi[2]$ or $\phi[3]$. Note that we are omitting the time index for brevity. Once the incoming and the recurrent weights of $\phi[1]$ are learned, the learner freezes them and goes to the

next stage. In the 2nd stage, it learns the incoming weights of $\phi[2]$, which can use both $\boldsymbol{x}$ and $\phi[1]$ as its inputs. The outgoing weight of $\phi[1]$—$w[1]$—is not fixed and continues to be updated. Similarly, in the 3rd stage, both $\phi[1]$ and $\phi[2]$ are frozen and fed to $\phi[3]$ as input. In each stage, the newly introduced feature can be connected to all prior features.

In this staged learning approach, the learner never learns more than one feature at a time. As a result, the effective size of the feature vector that is being learned is just one, and RTRL can be applied cheaply. In fact, since only a small subset of the network is being learned at any given time, constructive networks use even less per-step computation than columnar networks. They introduce one additional hyperparameter—steps-per-stage—that controls the number of steps after which the learner moves from one stage to the next.

Constructive networks are similar to recurrent cascade correlation networks (Fahlman, 1990). The main differences are that (1) cascade correlation networks learn new recurrent units by maximizing correlation with the error whereas constructive networks use the gradient w.r.t the prediction error, and (2) cascade correlation networks learn on a batch of data, whereas constructive networks learn from an online stream of data. The two differences are arguably minor. Rather, the bigger novelty is to combine constructive networks with columnar networks.

10.3 Columnar-Constructive Networks

Columnar-constructive networks (CCNs), as the name suggests, are a combination of columnar networks and constructive networks. In CCNs, we keep the multi-stage approach of the constructive networks; however, instead of learning a single feature in every stage, the learner learns multiple independent features.

A two-stage CCN is shown in Figure 10.2. In stage one, the learner learns the incoming weights of $\phi[1]$ and $\phi[2]$. Since $\phi[1]$ and $\phi[2]$ are independent of each other, they are equivalent to a columnar network with two features and can be learned efficiently together. In the second stage, the learner freezes

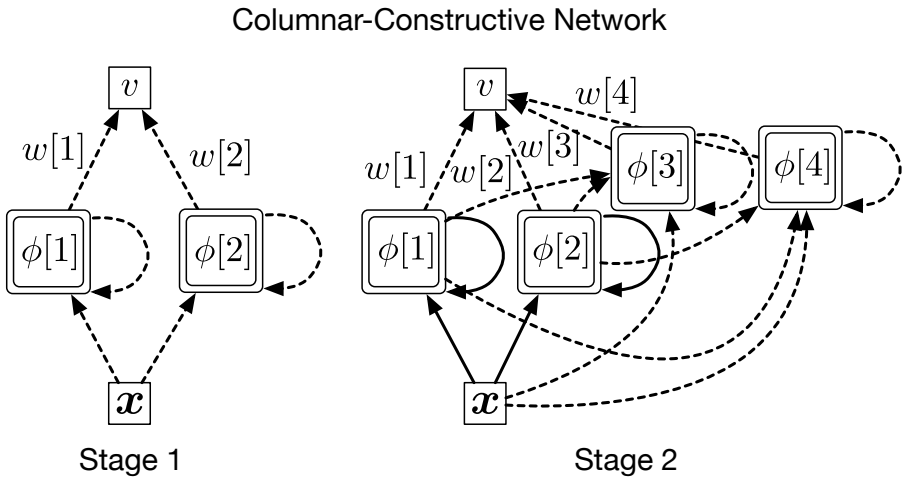

Figure 10.2: Columnar-constructive networks (CCNs) combine the ideas from Columnar and constructive networks. In each stage, they learn multiple features that are independent of each other, just like columnar networks. Across stages, they learn hierarchical features, similar to constructive networks.

the incoming and recurrent weights of $\phi[1]$ and $\phi[2]$ and learns the incoming weights of $\phi[3]$ and $\phi[4]$; the new features take both $\phi[1]$ and $\phi[2]$ as inputs. Once again, $\phi[3]$ and $\phi[4]$ are independent of each other and can be learned efficiently in parallel.

CCNs inherit the hyperparameters from columnar networks and constructive networks. Additionally, they have one new hyperparameter—features-per-stage—that controls the number of recurrent features learned in each stage.

10.4 The Animal Learning Benchmark

We evaluate the methods on the *trace patterning task* proposed by (Rafiee et al., 2022). It is an online prediction task that requires the learner to identify associations between patterns—conditional stimuli (CS)—that are predictive of future values of a cumulant—the unconditional stimuli (US). The goal is to predict the discounted sum of the US. Correct predictions require the ability to discriminate between patterns that lead to the US from those that do not. The time delay between the CS and the US necessitates retaining information from the past for making accurate predictions.

In our instantiation of trace patterning, the delay between the CS and

the US is uniformly randomly sampled to be between 24 and 36 steps after every CS and is called the inter-stimulus interval (ISI). The delay between the US and the next CS is uniformly randomly sampled to be between 80 and 120 steps after every US and is called the inter-trial interval (ITI). The CS consists of 6 features. When CS is present, three of the six features in the CS vector are one. Since $\binom{6}{3}$ is twenty, the CS vector can represent twenty different patterns. Ten randomly chosen patterns are followed by US=1 after ISI $\sim \mathcal{U}_{[24,36]}$ steps, whereas the remaining ten do not activate the US. Additionally, the observation vector has five random features that are not predictive of the US.

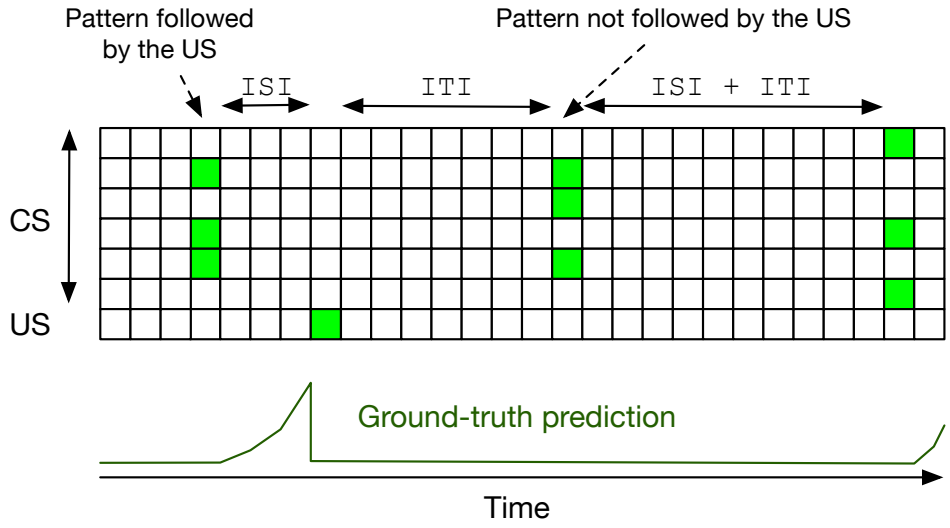

Figure 10.3: Visualization of the stream of experience for the trace patterning task. At each step, the learner receives an observation vector of length seven. The first six values are the CS and the last is the US. CS is either a vector of zeros or three of the six values are one. It can represent 20 different patterns. Ten of these patterns activate the US after ISI number of steps, whereas others do not. The learner has to predict the discounted sum of future values of the US. The bottom part of the figure shows the ground-truth prediction for the task.

A visual representation of experience from the trace patterning task without noisy components of the observation vector is shown in Figure 10.3. The vertical dimension shows the observations, and the horizontal dimension shows the time steps. At the fourth time step, three of the six features are one. After

three more time steps, the US becomes active. Then no components of the observation vector are active for ITI number of steps. After ITI steps, the CS again shows a pattern. The second pattern of the CS is not followed by the US. At the bottom of Figure 10.3, we show the ground truth return.

10.5 Experiments: Columnar-Constructive Networks on the Animal Learning Benchmark

We compared CCNs to fully connected RNNs learned by T-BPTT, columnar networks, and constructive networks. All comparisons used the LSTM cell architecture (Hochreiter & Schmidhuber, 1997) for recurrence. An important hyperparameter of T-BPTT is the truncation length parameter (k). In our experiments, all methods used the same amount of per-step computation. To keep the compute constant for T-BPTT for different values of k, learners using a larger truncation length parameter had fewer features.

Online Feature Normalization

A key to making our system work is online feature normalization. Unlike dense recurrent networks, features in our constructive and CCN networks can have varying numbers of incoming weights. This discrepancy can change the scale of each feature, making it hard to learn using a uniform step-size parameter. To address the varying scales, we use a simple form of online feature normalization. Our feature normalization is similar to an online version of batch normalization (Ioffe & Szegedy, 2015).

To normalize a feature, we maintain an online running estimate of its mean and variance. We then use the running estimates to normalize the feature to have zero mean and unit variance. Additionally, if the variance of a feature goes below a threshold, we set it to a small number ϵ, which is a hyperparameter. Given the unnormalized feature $\phi[j]$, the normalized feature $\hat{\phi}[j]$ is computed

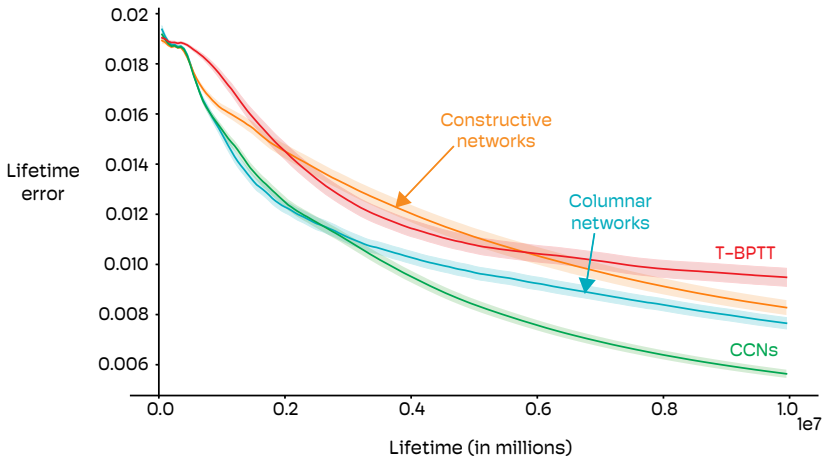

Figure 10.4: Performance of our algorithms and the best performing T-BPTT on the trace patterning task. All methods learned to make accurate predictions. Both columnar networks and constructive networks learned well, exceeding and matching the performance of the best T-BPTT. CCNs performed the best, showing that they combine the strengths of columnar networks and constructive networks. All plots are averaged over 100 seeds, and the shaded areas are +- standard error.

as:

$$\hat{\phi}_t[j] = \frac{\phi_t[j] - \mu_t[j]}{\max(\epsilon, \sigma_t[j])} \tag{10.10}$$
$$\text{where } \mu_t[j] = \mu_{t-1}[j]\beta + (1-\beta)\phi_t[j]$$
$$\sigma_t^2[j] = \sigma_{t-1}^2[j]\beta + (1-\beta)(\mu_t[j] - \phi_t[j])(\mu_{t-1}[j] - \phi_t[j]).$$

We set $\beta = 0.99999$ for all our experiments. $\mu_0[i]$ and $\sigma_0^2[i]$ are initialized to be 0 and 1 respectively, and ϵ is tuned; the values used for experiments in this chapter are in Table B.1.

Experimental setup

We used TD(λ) for learning with a per-step compute budget of $\approx$ 4,000 floating point operations. A single multiplication, addition, division, or subtraction is counted as an operation. All methods used $\lambda = 0.99$, $\gamma = 0.90$, and a lifetime of 10 million.

For each method, we individually tuned the step-size hyperparameter, ϵ, the steps-per-stage hyperparameter, the features-per-stage hyperparameter,

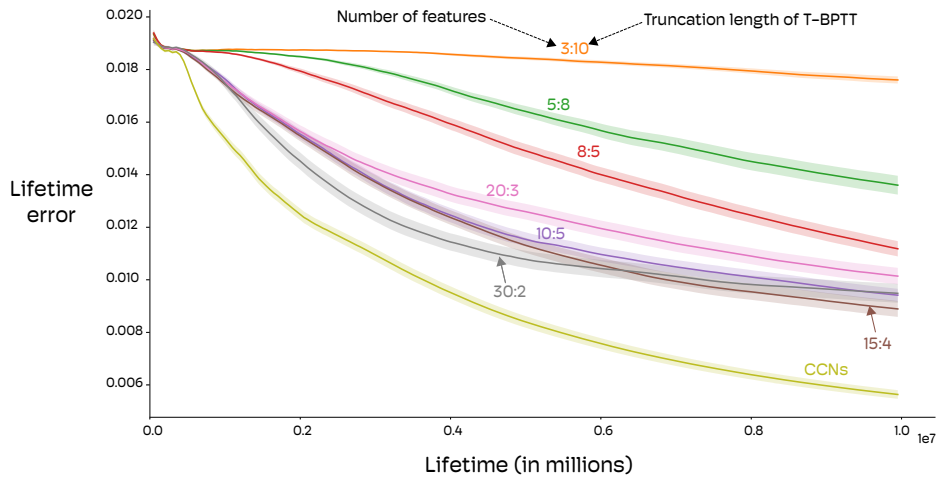

Figure 10.5: Different versions of T-BPTT on the trace patterning task. Each curve is denoted by two numbers: a:b. The first number indicates the truncation length parameter of T-BPTT, and the second number indicates the number of features in the learner. For example, 30:2 means an LSTM with two features trained with a truncation length parameter of 30. All versions use roughly the same amount of computation. We see that different values of truncation length parameters result in different performances. Large networks trained with small truncation length parameters—3:10 and 5:8—performed the worst. Smaller networks with larger truncation length parameters—15:4, 30:2, and 20:3—performed better. All lines are averaged over 100 random seeds.

and the truncation length hyperparameter. We report the results for the best-performing configuration. Details of hyperparameter tuning are in Appendix B.1. The columnar networks, constructive networks, and CCNs had 10, 5, and 16 features respectively. The number of features in constructive networks was dictated by the rate at which features were added. Because we only learned for 10 million time steps and set the steps-per-stage hyperparameter to 1 million, constructive networks ended up using significantly less compute than the allocated compute budget. T-BPTT used a truncation length parameter of 15 and had four features.

Results

We start by looking at the learning curves for all four methods in Figure 10.4. The three methods introduced by us learned to reduce the prediction error over time. Among our algorithms, constructive networks performed the worst.

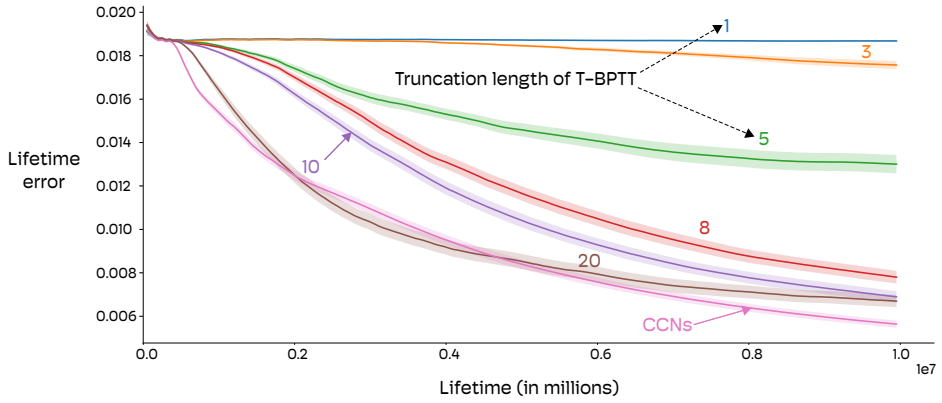

Figure 10.6: LSTMs with 10 features trained using truncation length parameters of 1, 3, 5, 8, 10, and 20. For each value of the truncation length parameter, we independently tuned the step-size parameter. As the truncation length increased, the performance improved at the expense of more computation. The sensitivity of performance to truncation length parameter highlights the impact of bias introduced by truncation. All lines are averaged over 100 random seeds and the shaded regions correspond to +- standard error.

All three methods outperformed the best T-BPTT, and CCNs performed the best.

We further investigated the sensitivity of T-BPTT to values of the truncation length parameter. We first considered the impact of reallocating resources, allowing T-BPTT to have more features trained with smaller truncation length parameters and vice-versa. We see from Figure 10.5 that when the truncation length parameter was much smaller than the longest dependency in the learning problem—36—the performance dropped significantly. T-BPTT performed the best when it selected a smaller network (four features) and a larger truncation parameter ($k = 15$).

We conducted another experiment where we allowed T-BPTT to use more computation than the allocated budget. We fixed the number of features to 10 and used different truncation length parameters. We report the results in Figure 10.6. Networks with the largest truncation length parameter—brown line—performed almost as well as CCNs. However, it used around seven times more per-step computation than CCNs.

Columnar-constructive networks are a promising solution for tuning recurrent features using gradients online. This chapter evaluates their performance

on the animal learning benchmark. Javed et al., (2023) evaluated them on the atari prediction benchmark and showed that they outperformed tuned T-BPTT baselines. The biggest limitation of CCNs is that they grow indefinitely and use more and more computation over time. One way to get past this limitation is to augment them with a method for removing features that are not useful for the task at hand.

Chapter 11

Conclusions and Future Work

In this dissertation, I presented computationally efficient algorithms for fast learning from online data streams. The motivation for quick and efficient learning is the big world hypothesis, which states that the world is orders of magnitude larger than the agent, and online continual learning using computationally efficient algorithms is necessary for achieving goals in big worlds.

The solution methods presented in earlier chapters are divided in two parts: fast and robust linear learning, and fast non-linear recurrent feature discovery. Algorithms proposed in both parts are computationally efficient and scalable to data streams that consist of large observation vectors.

A promising direction for future work is to combine the algorithms in Part I and Part II to develop a large scale system that continually generates new features by imprinting, learns with those features using SwiftTD, and adapts those features using columnar-constructive networks. It would be interesting to see its behavior when learning with billions of parameters.

Another interesting direction is to explore the impact of feature generation by imprinting and step-size optimization on the phenomenon of catastrophic forgetting. Imprinting provides an easy way to generate sparse features. Sparsity can help protect learned knowledge from being overwritten easily (see work by Liu et al., 2019 and Javed & White, 2019). Step-size optimization provides another venue for alleviating catastrophic forgetting by reducing the step-size parameters of some features and preserving knowledge learned with them. In some preliminary experiments not reported in this dissertation, I found that

both imprinting and SwiftTD alleviated the problem of catastrophic forgetting. A more systematic analysis is needed for a more definitive conclusion.

Some technical directions remain untouched. One direction is to extend SwiftTD to Actor-Critic algorithms by adding step-size optimization and update bounds to the policy parameters. The key challenge is deriving a bound, analogous to the η-bound, that restricts how much the policy changes from a single sample. Another is to develop princinpled algorithms for exploration in big worlds. Algorithms that aim to systematically explore the state space are intractable in big worlds. More scalable solutions are needed that use the feature vector to guide exploration.

Another direction is to design scalable solutions that can deal with correlated features. If the feature generation process is not careful, then it can generate many features that are highly correlated with each other. These features compete with each other and, even if they are useful, the high correlation makes it difficult for a single feature to get tenured status. In Chapter 9 I sidestepped this problem by limiting the generators to a discrete number of choices and using a global cache to prevent the generation of identical features. A more scalable solution is needed that works with generators that create an unbounded number of features, such as generators for differentiable recurrent cells.

References

Blalock, D., Gonzalez Ortiz, J. J., Frankle, J., & Guttag, J. (2020). What is the state of neural network pruning? In *Proceedings of Machine Learning and Systems*.

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*.

Berner, C., Brockman, G., Chan, B., Cheung, V., Debiak, P., Dennison, C., ... & Zhang, S. (2019). Dota 2 with large scale deep reinforcement learning. *arXiv preprint arXiv:1912.06680*.

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., ... & Amodei, D. (2020). Language models are few-shot learners. In *Advances in Neural Information Processing Systems*.

Dabney, W., & Barto, A. (2012). Adaptive step-size for online temporal difference learning. In *Proceedings of 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*.

Dohare, S., Hernandez-Garcia, J. F., Lan, Q., Rahman, P., Mahmood, A. R., & Sutton, R. S. (2024). Loss of plasticity in deep continual learning. *Nature*.

Dong, S., Van Roy, B., & Zhou, Z. (2022). Simple agent, complex environment: Efficient reinforcement learning with agent states. *Journal of Machine Learning Research*.

Evci, U., Gale, T., Menick, J., Castro, P. S., & Elsen, E. (2020, November). Rigging the lottery: Making all tickets winners. In *Proceedings of Inter-*

national Conference on Machine Learning.

Fahlman, S. (1990). The recurrent cascade-correlation architecture. In *Advances in Neural Information Processing Systems*.

French, R. M. (1999). Catastrophic forgetting in connectionist networks. *Trends in Cognitive Sciences*.

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]. Department of Computing Science, University of Alberta, Edmonton.

Han, S., Mao, H., & Dally, W. J. (2015). Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*.

He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*.

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

Ioffe, S., & Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *Proceedings of International Conference on Machine Learning*.

Javed, K., & White, M. (2019). Meta-learning representations for continual learning. In *Advances in Neural Information Processing Systems*.

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 Ma-*

chine Learning Research.

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

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

Kirkpatrick, J., Pascanu, R., Rabinowitz, N., Veness, J., Desjardins, G., Rusu, A. A., ... & Hadsell, R. (2017). Overcoming catastrophic forgetting in neural networks. In *Proceedings of the National Academy of Sciences*.

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. In *Advances in Neural Information Processing Systems*.

Kumar, S., Marklund, H., Rao, A., Zhu, Y., Jeon, H. J., Liu, Y., & Van Roy, B. (2023). Continual learning as computationally constrained reinforcement learning. *arXiv preprint arXiv:2307.04345*.

LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. *Nature*.

LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. In *Proceedings of the IEEE, 86(11), 2278-2324*.

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

Liu, V., Kumaraswamy, R., Le, L., & White, M. (2019, July). The utility of sparse representations for control in reinforcement learning. In *Proceedings of AAAI Conference on Artificial Intelligence*.

Mahmood, A. (2017). *Incremental Off-policy Reinforcement Learning Algorithms* [Doctoral dissertation]. Department of Computing Science, University of Alberta, Edmonton.

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

Mahmood, A. R., & Sutton, R. S. (2013). Representation search through generate and test. In *Workshop at AAAI Conference on Artificial Intelligence.*

Menick, J., Elsen, E., Evci, U., Osindero, S., Simonyan, K., & Graves, A. (2021). A practical sparse approximation for real time recurrent learning. In *Proceedings of 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.*

Mocanu, D. C., Mocanu, E., Stone, P., Nguyen, P. H., Gibescu, M., & Liotta, A. (2018). Scalable training of artificial neural networks with adaptive sparse connectivity inspired by network science. *Nature Communications.*

Mountcastle, V. B. (1957). Modality and topographic properties of single neurons of cat's somatic sensory cortex. *Journal of Neurophysiology.*

Rafiee, B., Abbas, Z., Ghiassian, S., Kumaraswamy, R., Sutton, R. S., Ludvig, E. A., & White, A. (2023). From eye-blinks to state construction: Diagnostic benchmarks for online representation learning. *Adaptive Behavior.*

Rummery, A., & Niranjan, M. (1994) *On-line Q-learning using Connectionist Systems* [Technical Report]. Department of Engineering, University of Cambridge.

Santurkar, S., Tsipras, D., Ilyas, A., & Madry, A. (2018). How does batch normalization help optimization? In *Advances in Neural Information Processing Systems.*

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

Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., ... & Hassabis, D. (2017). Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815.*

Silver, D., Sutton, R. S., & Müller, M. (2008). Sample-based learning and search with permanent and transient memories. In *Proceedings of Inter-*

national Conference on Machine Learning.

Sutton, R.S., Koop, A., & Silver, D. (2007). On the role of tracking in stationary environments. In *Proceedings of International Conference on Machine learning*.

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

Sutton, R. S., & Barto, A. (2018). *Reinforcement Learning: An Introduction* 2nd Edition. MIT Press.

Sutton, R. S. (1992). Adapting bias by gradient descent: An incremental version of delta-bar-delta. In *Proceedings of 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. In *Proceedings of International Conference on Autonomous Agents and Multiagent Systems*.

Sutton, R. S. (2020). *Are You Ready to Fully Embrace Approximation?* (June 8, 2020) [Video] (link).

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

Thill, M. (2015). *Temporal Difference Learning Methods with Automatic Stepsize 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*.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., ... & Polosukhin, I. (2017). Attention Is All You Need. In *Advances in Neural Information Processing Systems*.

Vinyals, O., Babuschkin, I., Czarnecki, W. M., Mathieu, M., Dudzik, A., Chung, J., ... & Silver, D. (2019). Grandmaster level in StarCraft II using multi-agent reinforcement learning. *Nature.*

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*.

Appendix A

Baseline Algorithms

Algorithm 16: TIDBD(λ) by Kearney et al. (2018)

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

Algorithm 17: Step-size Optimization for TD(λ) proposed by Thill (2015)

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

Algorithm 18: TD(λ) with Dabney & Barto's (2012) bound

Hyperparameters: $\boldsymbol{\alpha}, \lambda$
Initializations: $\boldsymbol{w} \leftarrow \mathbf{0} \in \mathbb{R}^n, \boldsymbol{z} \leftarrow \mathbf{0} \in \mathbb{R}^n, \boldsymbol{\phi^{old}} \leftarrow \mathbf{0} \in \mathbb{R}^n, v^{old} = 0$
while *alive* **do**
  Receive $\boldsymbol{\phi}$, γ, and r
  $v \leftarrow \sum_{\phi[i] \neq 0} w[i]\phi[i]$
  $\delta \leftarrow r + \gamma v - v^{old}$
  $b \leftarrow \sum_{i=0}^{n} \alpha_t[i]z[i]\left(\gamma\phi[i] - \phi^{old}[i]\right)$
  for $z_i \neq 0$ **do**
    $w[i] \leftarrow w[i] + \min\left(1, \frac{1}{b}\right)\alpha_t[i]\delta z[i]$;
    $z[i] \leftarrow \gamma\lambda z[i]$;
  for $\phi_i \neq 0$ **do**
    $z[i] \leftarrow z[i] + \phi[i]$
  $v^{old} \leftarrow \sum_{\phi[i] \neq 0} w[i]\phi[i]$
  $\boldsymbol{\phi^{old}} \leftarrow \boldsymbol{\phi}$

Appendix B

Hyperparameters

B.1 Columnar-Constructive Networks

The hyperparameters for columnar networks, constructive networks, columnar-constructive networks, and T-BPTT learners were tuned independently. For each hyperparameter configuration, we used five random seeds and looked at the average performance of all five seeds to pick the best hyperparameters. We then used the best hyperparameter configuration to run the experiments with 100 seeds. A list of the hyperparameters and their values are in Table B.1.

Hyperparameter	Hyperparameter values
Step-size	$1^{-2}, 3^{-3}, 1^{-3}$, $3^{-4}, \mathbf{1^{-4}}, 3^{-5}$
Adam parameters	0:0.9999:$1e^{-8}$
Discount factor	0.90
Eligibility trace decay rate	0.99
Truncation: Hidden features (T-BPTT)	2:13, 3:10, 5:8, 8:5, 10:5, **15:4**, 20:3, 30:2
Features-per-stage (CCN)	4
Steps-per-stage (CCN)	2.5 million
Steps-per-stage (Constructive)	1 million
Total steps	10 million
Seeds for parameter sweep	$\{0, 1, 2, 3, 4\}$
Seeds for best parameter configuration	$\{0, 1, \cdots, 99\}$
Min division term (CCNs and Constructive)	$\{0.01, \mathbf{0.001}\}$

Table B.1: Hyperparameter sweeps used for comparing columnar-constructive networks, columnar networks, constructive networks, and T-BPTT.

B.2 SwiftTD

For both SwiftTD and True Online TD(λ), we swept over their hyperparameters as shown in Table B.2. We used the same hyperparameters for both the linear learner and the last layer of the neural network learner. The experiments with LFA were deterministic and did not require multiple runs. The experiments with convolutional networks were stochastic due to the random initialization of the weights. For statistical significance, we did hyperparameter sweeps with 5 runs for each configuration. We then picked the best-performing configuration and did an additional 15 runs.

Symbol	Description	Algorithm	Values
α^{init}	Initial step-size parameter	SwiftTD	0.000001
α	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}	Step-size (kernels)	Both	$1e^{-1}, 1e^{-2}, 1e^{-3}, 1e^{-4}$,
θ	Meta step-size	SwiftTD	$1e^{-2}, 1e^{-3}, 1e^{-4}$
η	Max correction ratio	SwiftTD	0.3, 0.1
ϵ	Decay factor	SwiftTD	0.999, 0.99

Table B.2: Hyper-parameters used in the experiments of SwiftTD. 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).