Two-Timescale Networks for Nonlinear Value Function Approximation

by

Wesley Chung

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

Master of Science
in Statistical Machine Learning

Department of Computing Science

University of Alberta

Abstract

Policy evaluation, learning value functions, is an integral part of the reinforcement learning problem. In this thesis, I propose a neural network architecture, the Two-Timescale Network (TTN), for value function approximation which utilizes linear function approximation for the value function with learned features. By separating these two learning processes—approximating the value function and learning features—we can utilize classic policy evaluation methods suited for linear function approximation but still obtain nonlinear estimates of the value function. Additionally, the separation facilitates proving convergence guarantees for the value estimates. This thesis contains empirical investigations about the choice of linear policy evaluation algorithm, the choice of objective for feature-learning and also presents some experiments in the control setting. We find that TTNs perform competitively with other algorithms which train both the features and the value function estimates jointly. In particular, utilizing least-squares temporal difference methods seem to provide the largest benefit and eligibility traces can also be helpful for linear time TD algorithms. Overall, this thesis provides evidence that separating feature and value learning is a promising direction for nonlinear value function approximation.

Preface

Parts of this thesis appeared as a publication at the International Conference on Learning Representations (ICLR) 2019.

To my parents

For always supporting me in life.

We can only see a short distance ahead,
but we can see plenty there that needs to be done.

– Alan Turing, 1950.

Acknowledgements

First and foremost, I would like to thank my supervisor, Martha, for her continued support in all facets of life as a graduate student. I always felt that she was looking out for my best interests and, for that, I cannot thank her enough. I feel very fortunate to have had her as a supervisor throughout these years and, without a doubt, it has been one of the most important factors contributing to my enjoyment of my Master's studies. I would also like to thank Michael Bowling and James Wright for being on my thesis committee. Finally, thank you to all the members of the Amii and RLAI groups. I made many wonderful friends and had countless thought-provoking discussions—it's been great! (special thanks to #boba_guys and meow rangers!)

Thank you to NSERC and FRQNT for providing me with scholarships to support me throughout my studies.

Contents

1 Introduction **1**

2 Policy Evaluation Background **6**
2.1 Markov Decision Processes 6
2.2 Value Functions and Policy Evaluation 7
2.3 Optimizaton Objectives and Temporal Difference Methods . . 8
2.4 Linear Algorithms 11

3 Two-Timescale Networks **13**
3.1 Training Algorithm 14
3.2 Theory and Convergence 15

4 Online Policy Evaluation **20**
4.1 Setup and Environments 20
4.1.1 TTN Settings 21
4.1.2 Environments 21
4.2 TTN and Competitors 23
4.3 Linear Algorithms 26
4.4 Utility of optimizing MSPBE 31
4.5 Surrogate Objectives 34

5 Control **37**

6 Conclusion and Future Directions **41**
6.1 Contributions 41
6.2 Future Directions 42

References **44**

List of Figures

3.1 Two-Timescale Network architecture 14

4.1 Learning curves of TTN and competitor algorithms 25
4.2 Learning curves for least-squares methods 28
4.3 Sensitivity curves for λ 29
4.4 Step size sensitivity curves 30
4.5 Comparison of the MSPBE and MSTDE 33
4.6 Learning curves for surrogate losses 35

5.1 Learning curves in control 39

Chapter 1

Introduction

The goal of reinforcement learning agents is to collect the maximal cumulative amount of reward. Value functions—the expected discounted sum of rewards in the future from a state—therefore play a central role in reinforcement learning, with many different uses. Given the optimal value function, an RL agent need only choose the action with the highest value to assure reward-maximizing behaviour. These optimal value functions can be learned from a process called generalized policy iteration, which alternates between two steps: one, learning the value function for the current policy and, two, improving the policy to lead the agent to higher value states. Value functions are also a crucial part of actor-critic algorithms, which directly learn a good policy. In these algorithms, the value function helps the agent differentiate between good and bad actions by comparing their performance to a baseline value. As a final example, value functions can be interpreted more generally as a form of predictive knowledge. By summarizing sums of future quantities, the agent has immediate access to predictions about the future—information that could be valuable for decision-making [13, 30, 38]. In sum, due to their pervasiveness in reinforcement learning, the task of learning these value functions—policy evaluation—has great importance.

In practice, policy evaluation can only ever be done approximately. In most domains of interest, the number of states the agent may encounter is far greater than the amount of memory available to the agent. As such, a tabular representation with one value per state would be infeasible. Instead,

we can transform each state into a feature representation and utilize a learnable parametric function to approximate the value function. By learning the parameters for an approximate value function, we can generalize across states and still provide reasonable estimates of any state's value.

The most well-studied function approximators are linear functions, i.e., the value of a state is given by the dot product of state features and a set of parameters. This class of functions is simple enough to enable the derivation of many efficient optimization algorithms for learning the parameters and proving convergence guarantees for them [6, 36, 37]. Yet, despite their simplicity, linear functions can serve as good approximations to the value function due to the flexibility of the features. In fact, there are few restrictions on the function mapping the states to features. For the appropriate set of features, a linear function of those features can approximate the true value function well—exactly even. Hence, for a given application, the practitioner's task is to design these features to obtain suitable performance. The quality of these features has a large impact on the effectiveness of the approximate value function.

Designing an appropriate set of features can be difficult and often requires expert knowledge of the domain at hand. This can place a large burden on the practitioner to engineer features. With the advent of deep learning, it has been found that good feature representations can often be learned from basic ones using neural networks [26]. This approach is appealing since it requires less domain knowledge and, by using a more expressive class of functions, can often achieve better performance than hand-engineered solutions [34].

Unfortunately, training neural networks poses its own set of challenges and the optimization problem is much more difficult than that of linear functions. In particular, for learning value functions, the semi-gradient TD algorithm would be an obvious choice, but this algorithm does not have any convergence guarantees with nonlinear function approximation. In fact, there are certain cases where it is known to diverge [40]. Other than the theoretical issues, in practice, deep RL algorithms often rely on many heuristic tricks to stabilize training, such as reward and gradient clipping, without which effective training is difficult or impossible. Some progress has made to tackle these issues

although they are still not yet solved.

In this work, we explore an algorithm to take advantage of both the capacity of neural networks to learn effective features but also the plethora of efficient optimization algorithms available to linear function approximation. The Two-Timescale Network (TTN) is an architecture in which we use two concurrent processes to obtain nonlinear estimates of the value function. The first consists of optimizing weights for linear function approximation and the second learns the features by optimizing a surrogate objective. This separation allows us to use algorithms for linear functions while retaining *nonlinear* estimates of the value function. Additionally, we are able to prove convergence guarantees for all the parameters by appealing to a two-timescale argument: treating the features as slow-changing and the value weights as fast-changing. In this way, the features can be treated as fixed from the perspective of the fast optimization process and we can utilize the convergence guarantees for the linear TD algorithms with fixed features. Note that the agent still improves the feature representation (slowly) over time.

Similar approaches have previously been explored for policy evaluation, learning features in tandem to a linear value function, though without the key aspect of TTNs—the separation of losses for value and feature learning. This division enables simpler optimization objectives to be used for each part and hence simplifies the overall algorithm. In particular, the (linear) mean-squared projected Bellman error (MSPBE) can be used for efficiently learning values while another objective is used for updating the features with stochastic gradient descent, avoiding the complexities of the nonlinear MSPBE. Other feature-learning approaches utilizing two-timescales have been explored, although they used the same objective for learning both values and features [11, 25, 27]. Yu et al. [43] provided algorithms for basis adaptation using other losses, such as Bellman error using Monte carlo samples, taking derivatives through fixed point solutions for the value function. Levine et al. [21] periodically compute a closed form least-squares solution for the last layer of neural network to assist training with stochastic gradient descent, with a Bayesian regularization term to prevent too much change. These prior meth-

ods were more complex since they did not explicitly separate the feature and value learning. The core idea of using two different objectives, one to drive the representation and one to learn the values, has not been thoroughly explored.

In practice, while the underlying idea behind TNNs is more general and could potentially be applied to any feature-learning mechanism provided there is a fast/slow dichotomy, we focus on learning features with neural networks. Concretely, we consider training a neural network on a surrogate objective using stochastic gradient descent algorithms and treating the final hidden layer as a set of features. These features can then be used with value-learning algorithms suited for linear functions while they are being learned. By choosing neural networks for the feature representation, we can readily make comparisons to typical approaches using end-to-end training of the whole network. In this case, the main distinction is that the value updates within TTNs do not affect the features even though values and features are learned concurrently.

In this thesis, we provide an investigation into TTNs and the various design choices associated with them. The contributions are as follows:

- We conduct a suite of experiments in the online policy evaluation setting and find that TTNs can perform competitively with other value-learning algorithms. Additionally, our investigations indicate that least-squares algorithms are the best choice for the linear value-learning head, with faster convergence and lower asymptotic error. Eligibility traces can also bring benefits to some of the linear algorithms. Concerning surrogate objectives, we find no clear consensus as to which is more effective for feature-learning, with varying results depending on the environment.

- In the control setting, we find that TTNs equipped with linear fitted-Q iteration for the value-learning can be much more effective than DQN. Notably, TTN is able to learn much quicker in the early phases of training and avoid the initial 'warmup' period where DQN suffers from low returns.

- We present a convergence result for TTNs with TD(λ), showing that this

strategy is indeed a theoretically-sound approach to learning nonlinear estimates of the value function under reasonable assumptions.

This thesis is organized into 6 chapters. Chapter 2 discusses some relevant background concepts. Chapter 3 introduces Two-Timescale Networks. Chapter 4 contains the results of our empirical investigations into TTNs in the policy evaluation setting. Chapter 5 showcases our results in the control setting. Finally, chapter 6 concludes the thesis with a discussion of possible future work.

Chapter 2

Policy Evaluation Background

In this chapter, we provide the reader with a brief review of the necessary background concepts including Markov Decision Processes, value functions and temporal difference learning. The relevant notation is also introduced.

2.1 Markov Decision Processes

The standard formalism in reinforcement learning is a Markov Decision Process (MDP). A MDP is defined as a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{P}, \gamma)$ where $\mathcal{S}$ is the set of states, $\mathcal{A}$ is the set of actions, $\mathcal{R} : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is the reward function, $\mathcal{P} : (\mathcal{S}, \mathcal{A}, \mathcal{S}) \to \mathbb{R}^+$ gives the probability of transitions and $\gamma : (\mathcal{S}, \mathcal{A}, \mathcal{S}) \to \mathbb{R}^+$ is the discount function.

The agent interacts with the environment through the following process. At time t, the agent is in some state S_t and it chooses an action A_t. Then, conditional on S_t and A_t, the environment returns the next state, the reward and a discount factor according to $\mathcal{P}$, $\mathcal{R}$ and γ respectively. This process is repeated indefinitely. Note that both episodic tasks and continuing tasks are covered in this framework by setting the discount function appropriately [42].

The agent chooses actions according to a *policy* $\pi : \mathcal{S} \times \mathcal{A} \to \mathbb{R}^+$, which produces the probabilities of picking each action in a given state. When interacting with the environment, the agent will choose actions by sampling according to π.

2.2 Value Functions and Policy Evaluation

Value functions are a central part of reinforcement learning. Formally, these are defined as the expected cumulative rewards into the future $V^\pi(s) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R_{t+1} | S_t = s\right]$ where R_t is a random variable corresponding to the reward at time t.

The task of policy evaluation consists of finding the value function for a given policy. This is a fundamental task in reinforcement learning since the value function represents the quality of the policy π as a measure of the amount of reward it would collect if it were run. Additionally, by considering an extension called General Value Functions (GVF) [38], we may be able to represent predictions about other quantities of interest other than the reward. Algorithms for learning value functions can also be used for learning GVFs due to their similarity. Aside from producing predictive knowledge, policy evaluation with rewards can be interleaved with policy improvement steps to produce better policies as part of generalized policy iteration [35].

In practice, it is usually not possible to expect to learn the value of every single state in $\mathcal{S}$. The size of $\mathcal{S}$ may be too large to expect to see every state even just a single time. In fact, most states will not be visited. In these cases, the agent must be able to generalize from the states it has observed to unseen states. To do this, we consider learning a parametrized approximation to the value function $v_{\boldsymbol{\theta}}(s)$, where $\boldsymbol{\theta}$ denotes the vector of parameters to be learned. If we choose a suitable form for the function approximator, then we can expect the agent to make good predictions about unseen states by generalizing from its past experiences.

A simple and effective choice of function approximator is a linear function. We let our prediction be $v_{\boldsymbol{\theta}}(s) = \boldsymbol{\theta}^\top \boldsymbol{x}(s)$, a linear combination of features of the state $\boldsymbol{x}(s)$. This is a well-studied class of functions, with many effective and theoretically-sound algorithms for policy evaluation existing already. The drawback of linear functions is that the accuracy of the approximation is heavily dependent on the features used, $\boldsymbol{x}$. Choosing appropriate features often requires specialized knowledge about the domain at hand and, in some

cases, may be difficult to specify manually even by experts.

Alternatively, it is possible to choose a flexible nonlinear function approximator such as neural networks. In this case, less expert knowledge may be needed to specify useful inputs and the neural network can better adapt to the given data. This comes at the cost of complicating the optimization problem, so it may be more difficult to find an adequate set of parameters. Specifically, more data may be required and sometimes training can completely fail, with no progress at all on the objective of interest. Furthermore, the theoretical aspects of neural networks are not well understood, making it challenging to obtain theoretical guarantees for algorithms.

2.3 Optimizaton Objectives and Temporal Difference Methods

Our end goal is to learn an accurate approximation of the value function $v_{\boldsymbol{\theta}}(s) \approx V^{\pi}(s)$ for all $s \in \mathcal{S}$. We can define an objective, the mean-squared value-error (MSVE) as

$$MSVE(\boldsymbol{v}) \stackrel{\text{def}}{=} \|V^{\pi} - v\|_{d_{\pi}}^2$$

where V^{π} is a vector with $|\mathcal{S}|$ entries corresponding to the true value function and d_{π} is the stationary distribution over states corresponding to the policy π. Unfortunately, we cannot directly optimize this objective since we do not have access to the true value function V^{π}. Instead, we can choose to optimize the mean-squared return-error (MSRE), which provides samples of the true value function.

$$MSRE(\boldsymbol{v}) \stackrel{\text{def}}{=} \sum_{n=1}^{N} (G_n - v(S_n))^2$$

where G_i is a sampled return, S_i is the corresponding state from which the return was calculated and N is the total number of sampled returns.

The drawback is that we need to wait until the end of an episode to be able to calculate the returns. For episodic tasks, this prevents the agent from learning within an episode and is not directly possible in continuing tasks. We can turn to temporal difference methods to avoid this problem and do

updates at every step. We consider three candidate objectives: the mean-squared Bellman error (MSBE), the mean-squared TD error (MSTDE) and the mean-squared projected Bellman error (MSPBE).

Several of these objectives are based on the Bellman equation, a special recurrence relation satisfied by value functions. The value function v associated to a policy π will satisfy the following equation:

$$v(s) = \sum_{s',r,a} \pi(a|s)p(s',r|s,a)(r(s,a,s') + \gamma(s,a,s')v(s'))) \text{ for all } s \in \mathcal{S}$$

We can write this equivalently using vector notation. First, define the Bellman operator $\mathcal{B} : \mathbb{R}^{|\mathcal{S}|} \mapsto \mathbb{R}^{|\mathcal{S}|}$ as mapping a vector $\mathbf{u} \in \mathbb{R}^{|\mathcal{S}|}$ to a vector $\mathbf{w}$ where the s-th entry of $\mathbf{w}$ is given by $\mathbf{w}_s = \sum_{s',r,a} \pi(a|s)p(s',r|s,a)(r(s,a,s') + \gamma(s,a,s')\mathbf{w}_{s'}))$. Then, the Bellman equation can be compactly rewritten as

$$\boldsymbol{v} = \mathcal{B}\boldsymbol{v}$$

The fixed point of the Bellman operator is the value function corresponding to the policy π.

Having defined the Bellman equation, the mean-squared Bellman error (MSBE) is a natural consideration since it captures the idea that the value function should satisfy this recurrence relation. Optimizing the MSBE directly tries to minimize the difference between the current state's value and the quantity given by the Bellman equation. Letting $\delta_t \stackrel{\text{def}}{=} R_{t+1} + \gamma_{t+1}v(S_{t+1}) - v(S_t)$, the TD error for one step, we can write this loss as

$$MSBE(\boldsymbol{v}) \stackrel{\text{def}}{=} \mathbb{E}\left[\mathbb{E}\left[\delta_t | S_t\right]^2\right] = ||\mathcal{B}\boldsymbol{v} - \boldsymbol{v}||^2_{d_\pi}$$

where the outer expectation is over the state distribution while the inner expectation is over the one-step transition dynamics. In the tabular case, we know that minimizing this objective would recover the true value function V^π so the MSBE seems like a reasonable choice. Unfortunately, it is not easy to minimize this objective in general. To obtain an estimate of the gradient. we would need to get two independent samples of the next state of a transition, which is impossible when an agent is interacting in a real environment (see chapter 11 of [35]).

A similar alternative which is easy to sample from is the mean-squared TD error:

$$MSTDE(\boldsymbol{v}) \stackrel{\text{def}}{=} \mathbb{E}\left[\delta_t^2\right]$$

where the expectation is over both the state distribution and the one-step rewards/transition dynamics. Minimizing this objective with stochastic gradient descent (differentiating with respect to the value function for both the current step and the next step) yields the residual gradient algorithm [2], which converges reliably (by gradient descent properties). For deterministic environments, this is also equivalent to minimizing the MSBE. The downside is that the residual gradient algorithms have empirically been found to converge slower than semi-gradient TD methods (when these converge) and asymptotically find worse estimates of the value function when used with function approximation [10, 35].

Finally, we have the MSPBE. To define this objective, we need to specify a set of value functions $\mathcal{V}$ representable by the function approximator of our choice. We can then write the loss as:

$$MSPBE(\boldsymbol{v}) \stackrel{\text{def}}{=} ||\Pi\mathcal{B}\boldsymbol{v} - \boldsymbol{v}||_{d_\pi}^2$$

where Π is a projection operator which maps a vector $\mathbf{w}$ to the nearest vector in $\mathcal{V}$ using the norm $||.||_{d_\pi}$ to measure distance.

This objective is perhaps the most natural in retrospect, since a solution of the MSPBE is the same as a TD-fixed point, a set of parameters for which semi-gradient TD would make no updates on average. For linear function approximation and some weak assumptions, there is a unique minimizer of the MSPBE. Similar to the MSBE, directly computing the gradient of the MSPBE yields an expression that requires two independent samples of the next state. Nevertheless, this problem can be circumvented by tracking appropriate quantities, as in the gradient TD methods. Unfortunately, an additional obstacle for the MSPBE is that the projection operator is not easy to compute for arbitrary function classes so, aside from linear function approximation, it can be difficult to optimize.

2.4 Linear Algorithms

In this section, we give descriptions of many of the algorithms of the TD family along with their update rules. Many of these algorithms were derived specifically to be used with linear function approximation and, as such, we later present experiments utilizing them in the TTN architecture. We describe the possible advantages that these linear TD algorithms may bring.

Gradient TD

The gradient TD family of algorithms, including TD with gradient corrections (TDC) [24] , was designed to offer more robust convergence guarantees. While the classic TD algorithm with linear function approximation may diverge in the off-policy case, TDC retains convergence guarantees. This is accomplished by modifying the classic TD update so it corresponds to the gradient of the mean-squared projected Bellman error. In this way, the fixed point of TDC is identical to that of TD but additionally, as a gradient descent method, convergence is assured.

The additional robustness does come at a cost since TDC requires an additional set of parameters to be estimated for the gradient correction term. Overall, factoring the additional updates, this means TDC is approximately twice as expensive in terms of time and memory as regular TD. Empirically, even in the on-policy case, TDC may offer certain advantages over regular TD such as lower sensitivity to step sizes.

Emphatic TD

Proposed by Sutton et al. [36], emphatic TD (ETD) was first motivated by the off-policy setting to find a convergent algorithm that could avoid the problems associated with the deadly triad of bootstrapping, off-policy updates and function approximation. While we focus on the on-policy setting in this work, ETD is nevertheless an interesting algorithm to try as it provides a new update rule which places more emphasis on certain states and de-emphasizes others. This emphasis mechanism results in a different fixed point compared to the classic TD algorithm and can potentially lead to better solutions.

True-online TD

The True-online TD (TOTD) algorithm [32] was motivated by the discrepancy between the forward and backward views of TD(λ) in the online setting, By introducing a new forward view, the authors find that there is a backward view that produces exactly the same set of updates, encapsulated in the TOTD algorithm. Experimental evidence showed this algorithm often performed better than vanilla TD(λ) for online learning [33].

Least-squares TD

Least-squares TD (LSTD) [6] trades off computational complexity, in time and memory, in order to achieve better sample efficiency. More specifically, LSTD requires the storage of the inverse of a $d \times d$ matrix and a vector of length d: $O(d^2)$ memory overall, where d is the number of features. By using the Sherman-Morrison formula, it is possible to incrementally update the matrix with one transition at a time at the cost of $O(d^2)$ computation per timestep. In summary, LSTD is d times more expensive than the previous linear algorithms in terms of both time and memory. In practice, LSTD has been found to have better sample complexity and be less sensitive to hyperparameters [6, 10].

A notable variant is forgetful LSTD (FLSTD). Introduced by Van Seijen et al. [41], FLSTD uses a modified update rule, allowing the estimates of A and b to focus on more recent transitions. This property can be advantageous if we are in a setting where older transitions are less reliable than recent ones, including control or, in the presence of changing features—as in Two-Timescale Networks.

Chapter 3

Two-Timescale Networks

In this chapter, we introduce the Two-Timescale Network (TTN) architecture, describing the learning algorithm along with its associated convergence guarantees. We also discuss possible surrogate objectives to be used within TTNs and associated desiderata.

TTNs split the task of learning an accurate value function into two parts: finding effective features and optimizing a linear value prediction based on them. The approximate value function is defined as $\hat{V}(s) \stackrel{\text{def}}{=} \boldsymbol{x}_{\boldsymbol{\theta}}(s)^\top \mathbf{w}$, where $\boldsymbol{\theta}$ represents the parameters of the feature construction and $\mathbf{w}$ are a set of linear weights. Essentially, the form is the same as for linear value function approximation. The only difference is that, in this case, the features are not fixed and are dependent on the parameters $\boldsymbol{\theta}$.

In practice, two learning processes are run in tandem; the agent updates both $\boldsymbol{\theta}$ and $\mathbf{w}$ at every step by each optimizing a separate objective. $\mathbf{w}$ is updated using any RL algorithm designed for linear function approximation, including TD, gradient TD or least-squares TD. Meanwhile, $\boldsymbol{\theta}$ is optimized by using stochastic gradient descent steps on some chosen surrogate objective L_{slow}.

In Fig 3.1, we see that we can interpret TTNs as a neural network with two heads. One outputs $\hat{V}(s)$, the value estimate and the other $\hat{Y}(s) = \boldsymbol{x}_{\boldsymbol{\theta}}(s)^\top \bar{\mathbf{w}}$, an auxiliary prediction required to compute the surrogate loss. For example, this could be another value estimate or a prediction of the next reward. Note that, here, there are other auxiliary parameters $\bar{\mathbf{w}}$ which do not affect the

features directly but assist in training them.

3.1 Training Algorithm

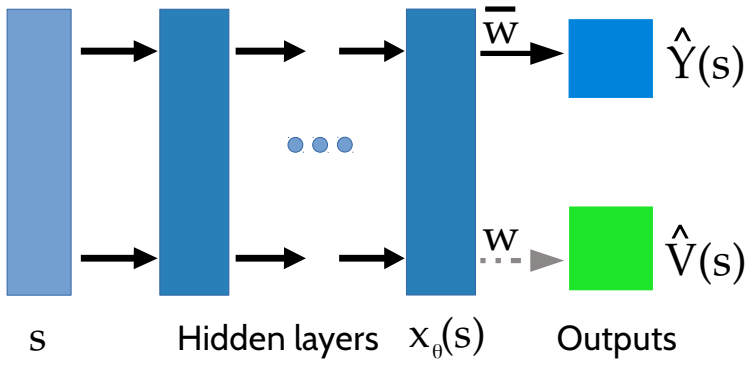


Figure 3.1: Two-Timescale Network architecture. The diagram depicts the neural network structure, starting with the input state as the leftmost layer. There are two output heads $\hat{Y}(s)$, for the surrogate objective, and $\hat{V}(s)$, for the value estimate. Note that the dotted line associated to w indicates that there is no error signal propagated backwards through this connection to update the previous layers since the value-learning process is separated from feature-learning.

Training TTNs are straightforward. After choosing a surrogate objective $\mathcal{L}_{slow}$, a neural network can be trained in the standard manner using stochastic gradient descent to minimize that objective. For example, we could choose $\mathcal{L}_{slow}(v) = MSTDE(v)$, the mean-squared TD error. For the value-learning process, we need to choose an appropriate algorithm for approximating the value function with linear function approximation—any algorithm from the TD family. The TD algorithm will be (implicitly) optimizing an objective $\mathcal{L}_{value}$. For every transition of experience the agent receives, we do an update for each process: one for the feature-learning neural network and one for the linear value-learning head. This is summarized in Algorithm 1.

While the algorithm is presented for the online case, it can be easily adapted to other settings. For example, a replay buffer can readily be incorporated by sampling transitions from a set of stored experiences. Additionally, mini-batches can also be utilized by adapting the learning algorithms appropriately.

Algorithm 1 Training of TTNs

procedure TRAIN($\mathbf{w}, \boldsymbol{\theta}, \bar{\mathbf{w}}, \pi$) ▷ π is a fixed policy
  Initialize $\boldsymbol{\theta}, \bar{\mathbf{w}}$ with Xavier initialization, $\mathbf{w}$ to 0 and the starting state s according to the environment
  while *training* **do**
    $a \leftarrow$ action chosen by π given s
    $r, s' \leftarrow$ Environment(s, a) ▷ Get reward and next state
    $\boldsymbol{\theta}, \bar{\mathbf{w}} \leftarrow$ GradientDescent on $\mathcal{L}_{slow}$ using sample (s, r, s')
    $\mathbf{w} \leftarrow$ Update on $\mathcal{L}_{value}$ using sample (s, r, s')
    $s \leftarrow s'$
  end while
  return learned parameters $\mathbf{w}, \boldsymbol{\theta}, \bar{\mathbf{w}}$
end procedure

For stochastic gradient descent-like methods, this can simply be accomplished by averaging the updates over the minibatch.

3.2 Theory and Convergence

Convergence results can be proved for the parameters of both the feature and value-learning processes. The intuition is as follows: The TD algorithms with linear function approximation can be proved to converge (under certain assumptions) [6, 37, 40]. These analyses treat the features as being fixed throughout learning. By using a two-timescale approach [3] and utilizing a smaller step size for the feature-learning part relative to the value-learning process, we can treat the features as slow-changing relative to the weights of the linear value function. In other words, we can consider the features to essentially be static, enabling us to use the regular convergence results for the TD algorithms.

A formal result is presented below:

Assumption 1: The pre-determined, deterministic, step-size sequence $\{\xi_t\}_{t\in\mathbb{N}}$ satisfies

$$\xi_t > 0 \; \forall t \in \mathbb{N}, \quad \sum_{t\in\mathbb{N}} \xi_t = \infty, \quad \sum_{t\in\mathbb{N}} \xi_t^2 < \infty.$$

These are the classic Robbins-Monro conditions on the step size to guarantee convergence. For example, $\alpha_t = \frac{1}{t^c}$ satisfies these conditions for $c \in (\frac{1}{2}, 1]$.

Assumption 2: The Markov chain induced by the given policy π is ergodic, *i.e.*, aperiodic and irreducible.

Assumption 2 implies that the underlying Markov chain is asymptotically stationary and henceforth it guarantees the existence of a unique steady-state distribution $\mathbf{d}_\pi$ over the state space $\mathcal{S}$ [20], *i.e.*, $\lim_{t\to\infty} \mathbb{P}(S_t = s) = \mathbf{d}_\pi(s)$, $\forall s \in \mathcal{S}$. Note that the policy π is unchanging throughout the whole process i.e. we are not performing and policy optimization.

Assumption 3: Given a realization of the transition dynamics of the MDP in the form of a sample trajectory $\mathcal{O}_\pi = \{S_0, A_0, R_1, S_1, A_1, R_2, S_2, \dots\}$, where the initial state $S_0 \in \mathcal{S}$ is chosen arbitrarily, while the action $\mathcal{A} \ni A_t \sim \pi(S_t, \cdot)$, the transitioned state $\mathcal{S} \ni S_{t+1} \sim P(S_t, A_t, \cdot)$ and the reward $\mathcal{R} \ni R_{t+1} = R(S_t, A_t, S_{t+1})$.

Assumption 3 ensures that the data is collected in the typical fashion for reinforcement learning agents, as a sequence of transitions.

Assumption 4-TD(λ): The pre-determined, deterministic, step-size sequence $\{\alpha_t\}_{t\in\mathbb{N}}$ satisfies:

$$\alpha_t > 0, \forall t \in \mathbb{N}, \quad \sum_{t\in\mathbb{N}} \alpha_t = \infty, \quad \sum_{t\in\mathbb{N}} \alpha_t^2 < \infty, \quad \lim_{t\to\infty} \frac{\xi_t}{\alpha_t} = 0.$$

This assumption encompasses the two timescales for the feature-learning and value-learning. With this condition, we can ensure that the value-learning is done at a significantly faster rate than the feature-learning. In other words, we can effectively treat the features as fixed when learning the linear weights of the approximate value function.

Theorem 1 *Let $\bar{\theta} = (\theta, \bar{\mathbf{w}})^\top$ and $\Theta \subset \mathbb{R}^{m+d}$ be a compact, convex subset with smooth boundary. Let the projection operator Γ^Θ be Frechet differentiable and $\widehat{\Gamma}^\Theta_{\bar{\theta}}(-\frac{1}{2}\nabla L_{slow})(\bar{\theta})$ be Lipschitz continuous. Also, let Assumptions 1-3 hold. Let $\mathcal{K}$ be the set of asymptotically stable equilibria of the following ODE contained inside Θ:*

$$\frac{d}{dt}\bar{\theta}(t) = \widehat{\Gamma}^\Theta_{\bar{\theta}(t)}(-\frac{1}{2}\nabla_{\bar{\theta}} L_{slow})(\bar{\theta}(t)), \quad \bar{\theta}(0) \in \mathring{\Theta} \text{ and } t \in \mathbb{R}_+.$$

Then the stochastic sequence $\{\bar{\theta}_t\}_{t\in\mathbb{N}}$ generated by the TTN converges almost surely to $\mathcal{K}$ (sample path dependent). Further,

TD(λ) Convergence: *Under the additional Assumption 4-TD(λ), we obtain the following result: For any $\lambda \in [0, 1]$, the stochastic sequence $\{\mathbf{w}_t\}_{t\in\mathbb{N}}$ generated by the TD(λ) algorithm within the TTN setting converges almost surely to the limit $\mathbf{w}^*$, where $\mathbf{w}^*$ satisfies*

$$\Pi_{\bar{\theta}^*} T^{(\lambda)}(\Phi_{\bar{\theta}^*}\mathbf{w}^*) = \Phi_{\bar{\theta}^*}\mathbf{w}^*, \tag{3.1}$$

with $\bar{\theta}^* \in \mathcal{K}$ (sample path dependent) and where $T^{(\lambda)}$ is the projected Bellman operator and $\bar{\theta}^*$ is a matrix with each row being the set of features for a state (with parameters $\bar{\theta}^*$).

The first part of this theorem states that the parameters of the neural network converge to a fixed point where the gradient is equal to 0. The exact convergence point will depend on the path taken by the iterates. Note that this does not preclude the possibility of converging to a saddle point of L_{slow}. While this remains a possibility for the nonconvex loss surfaces of neural networks, there is empirical and theoretical evidence that stochastic gradient descent can avoid saddle points and converge to a local minimum [14, 18].

The second part of the theorem states that the additional parameters of linear value function trained by TD(λ) also converge to a solution of the projected Bellman equation (ie. a minimum of the mean-squared projected Bellman error). For linear independent features, there would a unique solution but, since the features are generated by a neural network, this condition may not hold. As such, there could be infinitely many solutions satisfying the projected Bellman equation and the convergence point will also depend on the initialization and the exact path taken by the iterates.

While this theorem considers only TD(λ), similar convergence guarantees can be obtained for other TD algorithms such as least-squares TD and gradient TD. The proof of this theorem along with additional results can be found in the conference paper version of this thesis work [7].

Note that, while the value estimates depend on the features, there is no dependence in the other direction. Because of this unilateral coupling, it may

be possible to use a simpler approach to provide convergence guarantees. The two-timescale method was chosen due to its intuitive appeal—slow-changing features and fast-changing values—and since it is a technique that has been utilized in reinforcement learning previously [11, 37].

There are some noteworthy differences between the theory and the practical implementation of TTNs. While assumption 4 dictates that the step sizes of both learning processes must decay over time, in practice, we use constant learning rates. This matches conventional usage of stochastic gradient descent algorithms in reinforcement learning [1, 16, 26]. In this situation, while the convergence theorem would not hold as-is, we can expect the weights to converge to a region of low loss whose size depends on the step size used [4].

Furthermore, the two-timescale condition on the step sizes is not explicitly enforced in our experiments. Since the optimal step size is dependent on properties of the optimizer and the loss function, we instead tune the step sizes for each process empirically, without verifying that the step size for the feature parameters is lower than the one of the linear value weights. In our experiments, we happen to find that the step size for feature-learning is lower than the one for value-learning though this may be a spurious result and should not be taken too seriously due to differences in the optimization algorithms.

We can see that the update rules analysed here make use of a projection operator to a compact set, which is not present in the regular update. This is due to a technical condition for the proof requiring that the weights generated by the updates be bounded for the learning processes. Note that is a weak condition since the compact set can be very large and is allowed to grow over time, meaning it can eventually include any parameter value. Hence, this projection can have negligible or no impact so, in practice, we do not apply this projection for simplicity.

The theorem holds for twice continuously-differentiable functions. While this is a large class of functions, it does not include conventional neural networks with rectified linear activations (which were used in our experiments). To satisfy this requirement, we could utilize smooth versions of the rectified linear unit, such as the softplus function or the exponential linear unit [8].

Practically, we do not expect there to be a significant difference in performance between these choices.

Chapter 4

Online Policy Evaluation

In this chapter, we empirically investigate several aspects of Two-Timescale Networks, including the utility of different linear policy evaluation algorithms, the influence of the surrogate objective and comparisons to existing algorithms. These experiments were conducted on 6 different environments, with extensive hyperparameter tuning for all the algorithms.

4.1 Setup and Environments

Experiments were conducted in the online policy evaluation setting. The behaviour policy is fixed and the agent receives a single transition at every timestep according to its current state and the selected action, with which it can perform an update to its parameters. The agents were evaluated according to the mean-squared error (MSE) between the learned and true value functions for the fixed policy at certain checkpoints during training. For all plots, hyperparameters were chosen to minimize the average error across the checkpoints in the latter half of training.

The true value functions were estimated by sampling states using the behaviour policy and then running extensive rollouts from each of them using the behaviour policy until the episode terminated (for episodic tasks) or a cap was reached (for continuing tasks). The average return from those rollouts was treated as the true value function.

4.1.1 TTN Settings

For all the experiments, the TTN architecture had a single hidden layer of 256 hidden units and rectified linear activations to use as the representation, with the weights in the neural network initialized according to Xavier initialization [15]. The neural network was optimized using the AMSGrad optimizer [28] with $\beta_1 = 0$ and $\beta_2 = 0.99$ on the MSTDE (unless specified otherwise). For the linear head, the weights were all initialized to 0 with the optimization algorithm varying according to the experiment.

4.1.2 Environments

We used 6 environments to test our algorithms. These were puddle world, image and nonimage versions of catcher, acrobot, puck world and cartpole. The details are presented below:

Puddle World

This is is a classic environment introduced by Boyan and Moore [5] consisting of a 2-d continuous gridworld with two large puddles. The agent starts in the South-West corner and must reach the North-East corner, getting a large negative reward for crossing the puddles. The 2-dimensional state consists of the (x, y)-position of the agent.

The policy takes the North and East actions with equal probability, moving towards the goal.

Catcher

Catcher is a game from the Pygame Learning Environment [39]. Apples fall from the top of the screen and the agent must control a paddle at the bottom to catch them. The actions are left or right, each increases the velocity in the respective direction. A third "None" action does nothing. There are two versions of catcher, differing in their state. For the image version with visual inputs (I), the agent receives a 64×64 pixel grayscale frame showing the playing field. Since this image is not a Markov state due to the inability to perceive velocity of the paddle, we stack two consecutive frames and treat this stack as the state. For the nonimage version (NI), the agent receives a 4-

dimensional state vector consisting of the velocity of the paddle, the x-position of the paddle and the (x, y) positions of the apple.

The policy chooses the action in the direction of the apple if the apple is within a euclidean distance of 25 units of the paddle. If not, then 80% of the time, the agent chooses the "None" action and with 20% chance, an action is selected uniformly at random.

Puck World

Puck World is another game form the Pygame Learning Environment. In this game, the agent moves in a two-dimensional box towards a good puck while avoiding a bad puck. The 8-dimensional state consists of player x location, player y location, player x velocity, player y velocity, good puck x location, good puck y location, bad puck x location and bad puck y location. The action space consists of the 4 cardinal directions, each increasing the agent's velocity in that direction, along with a "None" action, doing nothing. The reward is the negative distance to the good puck plus a penalty of $-10 + x$ if the agent is within a certain radius of the bad puck, where $x \in [-2, 0]$ depends on the distance to the bad puck (the reward is slightly modified from the original game to make the value function less uniform over states).

The policy moves the agent towards the good puck, while having a soft cap on the agent's velocity. In more detail, to choose one action, it is defined by the following procedure: First, we choose some eligible actions. The None action is always eligible. The actions which move the agent towards the good puck are also eligible. For example, if the good puck is Northeast of the agent, the North and East actions are eligible. If the agent's velocity in a certain direction is above 30, then the action for that direction is no longer eligible. Finally, the agent picks uniformly at random from all eligible actions.

Acrobot

In the classic Acrobot domain, the agent consisting of two links has to swing up past a certain height. The agent observes a 4-dimensional state consisting of the angles and the angular velocities of each link. The available actions are three possible levels of torque to be applied to the joint.

The evaluated policy is obtained by training an agent with true-online

Sarsa on a tile coding representation and then fixing its learned epsilon-greedy policy.

Cart Pole

In the classic Cartpole environment, the agent has to balance a pole on a cart. The state is given by vector of 4 numbers: cart position, cart velocity, pole angle and pole velocity. The two available actions are applying a force towards the left or the right. Rewards are +1 at every timestep and an episode terminates once the pole dips below a certain angle or the cart moves too far from the center of the field.

The policy to be evaluated consists of applying force in the direction the pole is moving with probability 0.9 (stabilizing the pole) or applying force in the direction of the cart's velocity with probability 0.1. We chose this policy to ensure that the agent doesn't perform overly well, which would result in an uninteresting value function that is mostly constant across the state space.

4.2 TTN and Competitors

Here, we compare TTN to some baseline algorithms for online policy evaluation. We utilize TTN with least-squares TD (LSTD) as the value-learning algorithm since it seemed to perform best overall.

Nonlinear TD

This corresponds to the most simple and common approach of training a neural network, end-to-end with semi-gradient TD. This algorithm also incorporates a tunable λ parameter for eligibility traces. Note that this approach has no theoretical guarantees. It is known that semi-gradient TD can diverge with nonlinear function approximation in certain cases [40] but, in practice, it can often work well with neural networks.

Nonlinear TD-Reg

This variant of nonlinear TD is an adaptation of the LS-DQN algorithm [21] to online policy evaluation. In this algorithm, we incrementally estimate the LSTD solution for the last layer's weights, treating the final hidden layer as a set of features. Then, at every step, we compute the regular nonlinear TD

update with some additional regularization towards the LSTD solution. By adding this regularization term, we might expect to gain some of the benefits that LSTD enjoys in the linear setting, while still enabling the use of neural networks. The strength of this regularization is a tunable parameter that we sweep over.

Nonlinear GTD

This algorithm was derived by Maei et al. in [23] as an extension of the gradient TD algorithm to nonlinear function approximation. The main motivation was to derive a convergent algorithm for nonlinear function approximation. The authors consider a nonlinear MSPBE objective and develop an algorithm that performs gradient descent on it, using only linear time and memory in the number of parameters. Note that the fixed points of nonlinear TD are also fixed points of nonlinear GTD, so we could expect nonlinear GTD to find similar weights of similar quality (with additional stability). While this is the first theoretically-sound TD algorithm for nonlinear function approximation, it did not see much use and empirical evaluations of the algorithm are scarce.

ABTD and ABBE

These algorithms were introduced by Di Castro et al. [11]. Both of these algorithms function similarly. We use a regular fully-connected neural network and split the parameters into two sets: the last set of linear weights and the rest. During training, we set two different learning rates, one for each set. ABTD uses the semi-gradient TD update, while ABBE uses the full gradient of the MSTDE to compute updates. Note that we also utilize the AMSGrad optimizer.

This approach is similar to TTN, but with a key difference. The two set of weights are jointly trained as part of the same neural network. This is unlike TTN, where the optimization process for the weights of linear value function is separate from the one used to learn the features. In this architecture, there is only a single output head for the network.

By using a two-timescale approach, the authors are able to prove convergence of the parameters under standard assumptions, making this algorithm theoretically-sound.

Note that there is a third algorithm presented by Di Castro et al. in this paper, ABPBE, which optimizes the projected Bellman error. We did not run that algorithm since it was computationally infeasible, requiring memory proportional to $O(d^2m)$, where d is the number of features and m is the number of parameters of the features (this is large for neural networks). Also, the derivation was similar in spirit to that of nonlinear GTD, which we do include as a baseline. Hence, we omitted that algorithm.

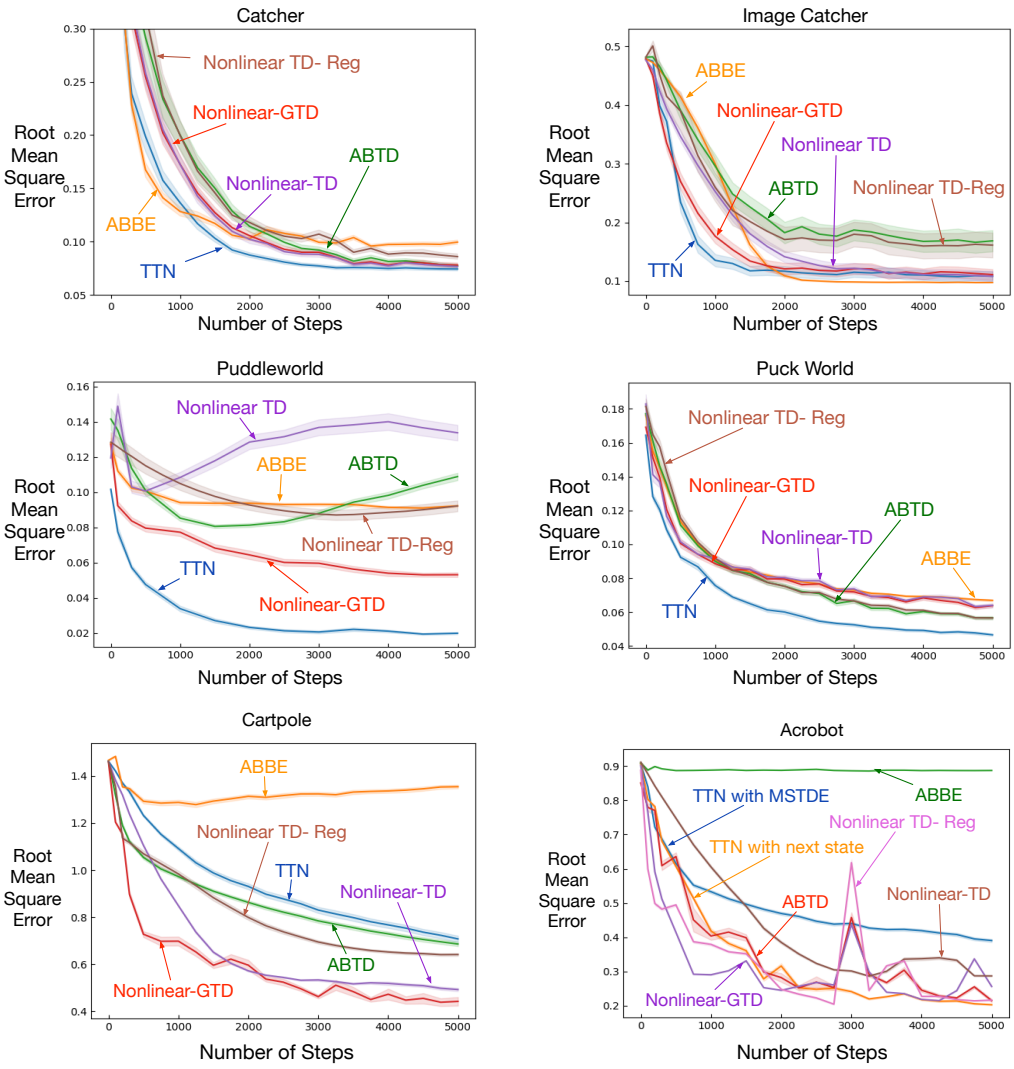


Figure 4.1: Learning curves of TTN and competitor algorithms. TTN offers competitive performance with the other algorithms across this set of environments.

From Fig 4.1, we find that TTNs is a competitive algorithm in most environments and, for some, performs best at all points during training. This difference is particularly visible in the Puddle World domain, where TTN achieves an asymptotic error of less than half the second best algorithm's. In the physical environments, Cartpole and Acrobot, TTN does perform worse compared to the others. Looking at the surrogate objective section, we see that this can be explained by the poor performance of the MSTDE to drive feature-learning. For Cartpole, using the semi-gradient MSTDE instead yields much better results while, for Acrobot, the next state surrogate objective is adequate.

These results also show that nonlinear TD, the most common algorithm, does not perform as well as many of the alternatives. On the other hand, nonlinear GTD, which adjusts nonlinear TD to be theoretically-sound, outperforms vanilla nonlinear TD in every domain and is, overall, quite effective compared to all the other competitors.

4.3 Linear Algorithms

We investigate the differences between choices of learning algorithms for the linear value head with particular attention to least-squares methods and eligibility traces. We consider the following algorithms: classic TD (TD), TD with gradient corrections (TDC), emphatic TD (ETD), true-online TD (TOTD), true-online emphatic TD (TOETD), least-squares TD (LSTD) and forgetful least-squares TD (FLSTD). Aside from TD and TDC, these methods have been limited to linear function approximation, without extensions to nonlinear functions such as neural networks.

Least-squares methods have been found to achieve better sample efficiency compared to other linear methods [6, 19]. The main drawback is their computational and memory costs, requiring $O(d^2)$ time and memory where d is the number of features, as opposed to the linear cost for the other algorithms. While problematic when we must deal with very large number of features, this cost can be acceptable when d is relatively low. TTNs provide a good use

case for least-squares algorithms since the user can decide how large to make the hidden layers. Additionally, computing a forward pass through a fully-connected layer in a neural network already requires $O(d^2)$ time and memory, so employing a least-square method does not increase the (asymptotic) total cost.

From Fig. 4.2, we see that the least-squares methods perform significantly better than other linear algorithms, confirming previous findings about LSTD's sample efficiency. There doesn't seem to be a large difference between LSTD and FLSTD. This is a bit surprising considering that LSTD simply computes an average over all past features (and rewards), including invalid features from older transitions. As such, we would expect FLSTD to have an advantage as it can put more weight on the recent features, which correspond better to the current feature parameters. To explain LSTD's good performance, it is possible that the η hyperparameter (for initializing the A matrix) can prevent the early transitions from having an unduly large impact on the average A matrix (estimated by LSTD) by regularizing it towards a diagonal matrix. At the later stages of training, A can still be accurate as the features are then more stable and the influence of the early, invalid features have been diluted by averaging across a large number of transitions.

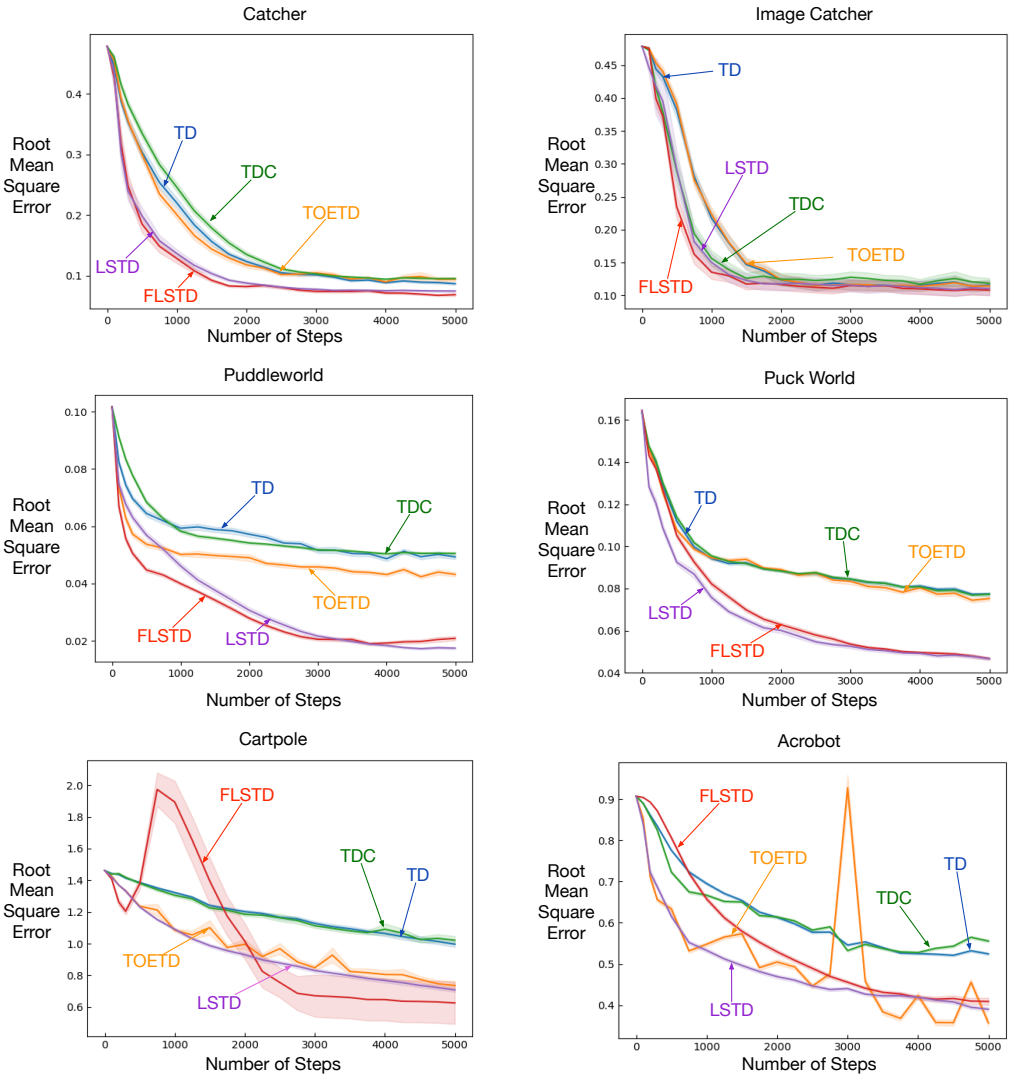


Figure 4.2: Learning curves for least-squares methods. These plots compare the performance of LSTD and FLSTD to some of the other linear algorithms. Both of the least-squares methods displayed better performance in terms of learning speed and asymptotic error in all the domains, with substantial differences in certain environments.

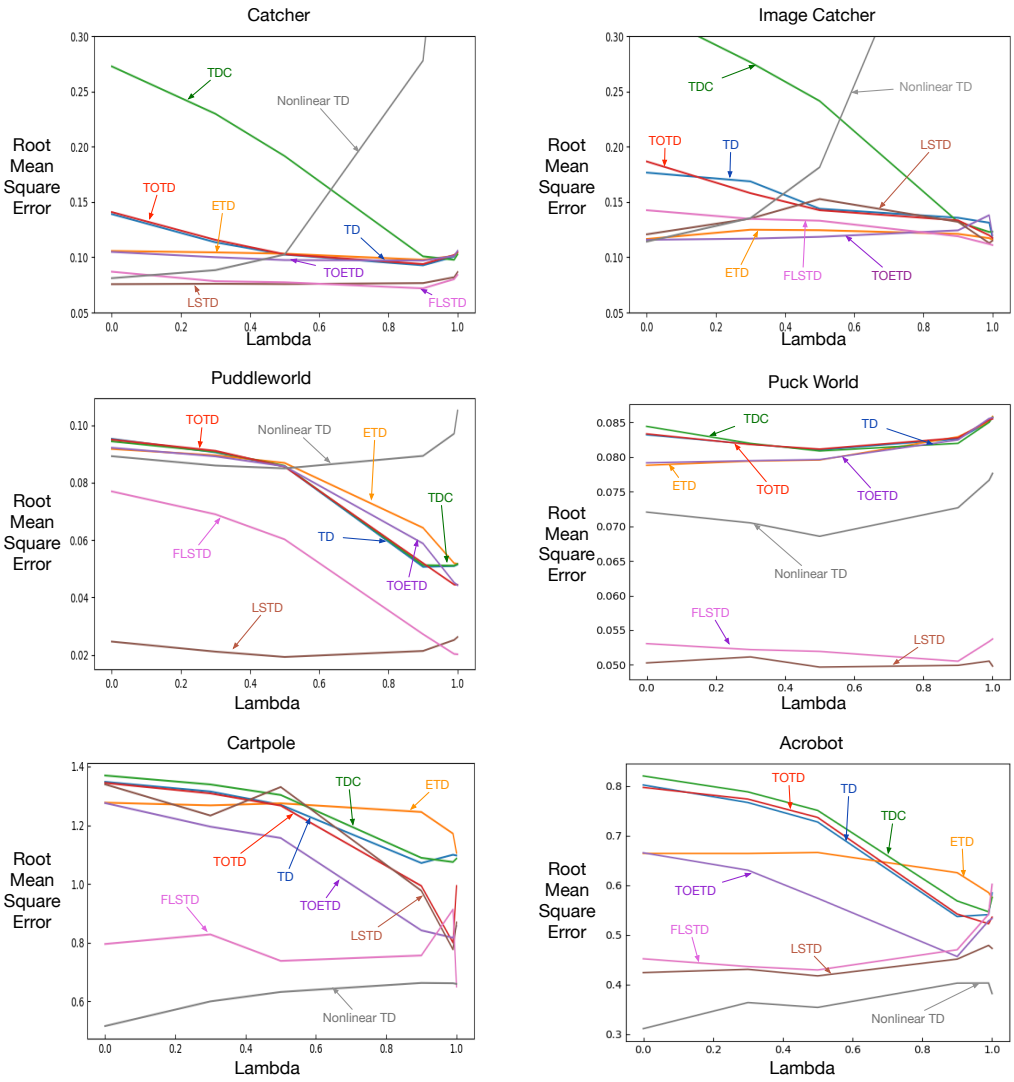


Figure 4.3: Sensitivity curves for λ. The average RMSE over the last half of training is plotted against the value of λ. Within TTN, aside from the least-squares methods, the best-performing values of λ are closer to 1. On the other hand, for nonlinear TD, $\lambda = 0$ is the best choice.

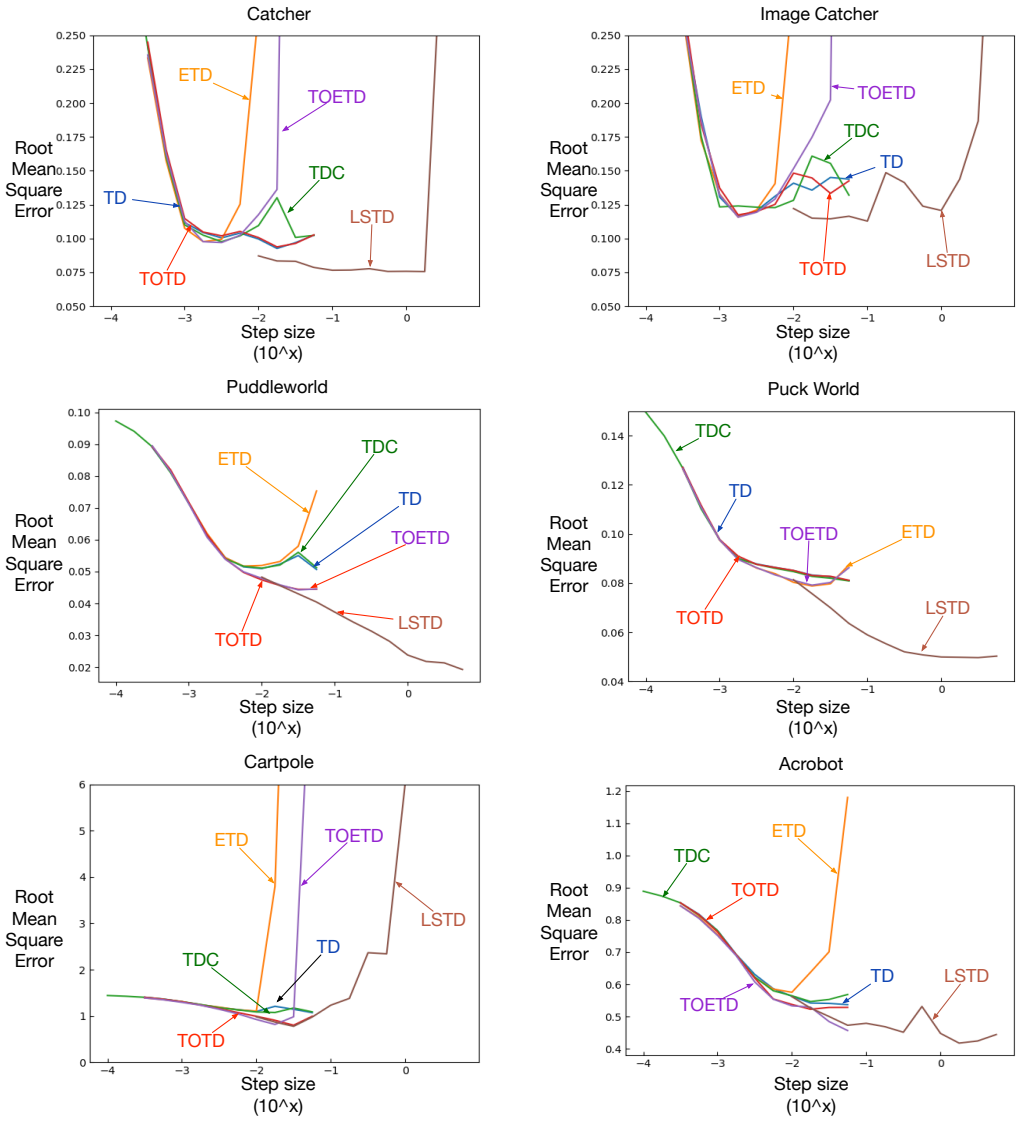


Figure 4.4: Step size sensitivity curves. The average RMSE over the last half of training is plotted against the value of α. The least-squares methods seem to be the least sensitive to their (effective) step size parameter, while the other linear algorithms seem to have similar shapes. Out of those, TOTD seems to be slightly more robust to the step size.

Eligibility traces have been found to speed up learning by spreading credit backwards in time and updating previously seen states. Their derivation hinges on an equivalence between the forward and backward views of TD updates with the λ-return target. This enables the update to be done in an online, incremental fashion without having to wait to the end of an episode. Unfortu-

nately, this equivalence only holds for linear function approximation, without a theoretically-sound extension to more general functions. Nevertheless, by matching the form of the eligibility trace update rule, it is still possible to write a trace update for nonlinear function approximators. So, for nonlinear TD(λ) and a generic function approximator with parameters $\boldsymbol{\theta}$, we would have

$$\mathbf{e}_{t+1} = \gamma\lambda\mathbf{e}_t + \nabla_{\boldsymbol{\theta}} V_{\boldsymbol{\theta}}(S_t)$$

where $\mathbf{e}_t$ is the eligibility trace vector and $\nabla_{\boldsymbol{\theta}} V_{\boldsymbol{\theta}}(S_t)$ is the gradient of the value function with respect to the parameters.

From the λ sensitivity plots (Fig. 4.3), we can observe that most of the linear algorithms benefit from a high value of λ, with the exception of the least-squares methods. LSTD and FLSTD are relatively insensitive to the choice of λ. To explain the difference between least-squares methods and the rest, we provide the following hypothesis: the main purpose of eligibility traces is to help temporal credit assignment. Least-squares methods are able to compute a batch solution on all previously seen data (in an incremental manner). By doing so, temporal credit assignment is already done effectively on all previous states and rewards, obviating the need for traces. Note that increasing λ does still have an effect on the fixed point solution (moving it closer to the Monte-Carlo solution), which could explain the slight differences in performances across λ values.

Contrary to the linear algorithms, nonlinear TD's performance suffers from any $\lambda > 0$. This is unsurprising considering that the naive eligibility trace used is not theoretically-sound. The addition of these traces seems to be hindering learning instead of assisting it.

4.4 Utility of optimizing MSPBE

In this section, we highlight the utility of optimizing the MSPBE as opposed to the MSTDE for learning values. More specifically, one, we show that training a neural network end-to-end on the MSTDE provides worst estimates of the value function than only using the MSTDE to drive feature-learning along with

optimizing the (linear) MSPBE for values. Second, we show that optimizing MSTDE indeed yields better features by comparing the TTN's performance to one where the features are fixed (a neural network at initialization). For completeness, we also include a curve for using the MSTDE only on the linear part with fixed features.

For the end-to-end training on the MSTDE, we utilize the AMSGrad optimizer and for the TTN linear part, we use LSTD (which optimizes the MSPBE). These choices were made to reflect what we believe to be are good, realistic settings for these algorithms in practice. Other settings were the same as before.

To summarize, we test these four settings:

- MSPBE-TTN: LSTD on linear part, MSTDE for feature learning. This is the regular TTN algorithm.
- MSPBE-Fixed: LSTD on linear part, no feature learning. We test what happens when the features are fixed, instead of learned. We want to check that learning features does provide some benefit.
- MSTDE: MSTDE from end-to-end, i.e. MSTDE for linear part and MSTDE for feature learning. We test what happens if we do not use the MSPBE for the linear part. We want to check that using the MSPBE for learning the value function is superior to using the MSTDE.
- MSTDE-Fixed: MSTDE for linear part, no feature learning. We test what happens if we only use the MSTDE with a fixed representation. This is a sanity check, we expect this to be the worse setting.

From these plots (Fig. 4.5), we see that the two aforementioned phenomena do occur. There is some variability between the results depending on the specific environment. In certain environments, optimizing the MSTDE directly seems to be relatively effective such as Catcher but, on others, the MSPBE is a better choice. Note that in none of the environments do we see that optimizing the MSPBE with TTN gives worse results than the end-to-end training on the MSTDE.

These results are in line with previous work [31] that find that the MSTDE may not be the best objective to optimize directly. On other hand, it has been argued that the MSPBE is the most appropriate choice of objective [35] and our experiments support this claim.

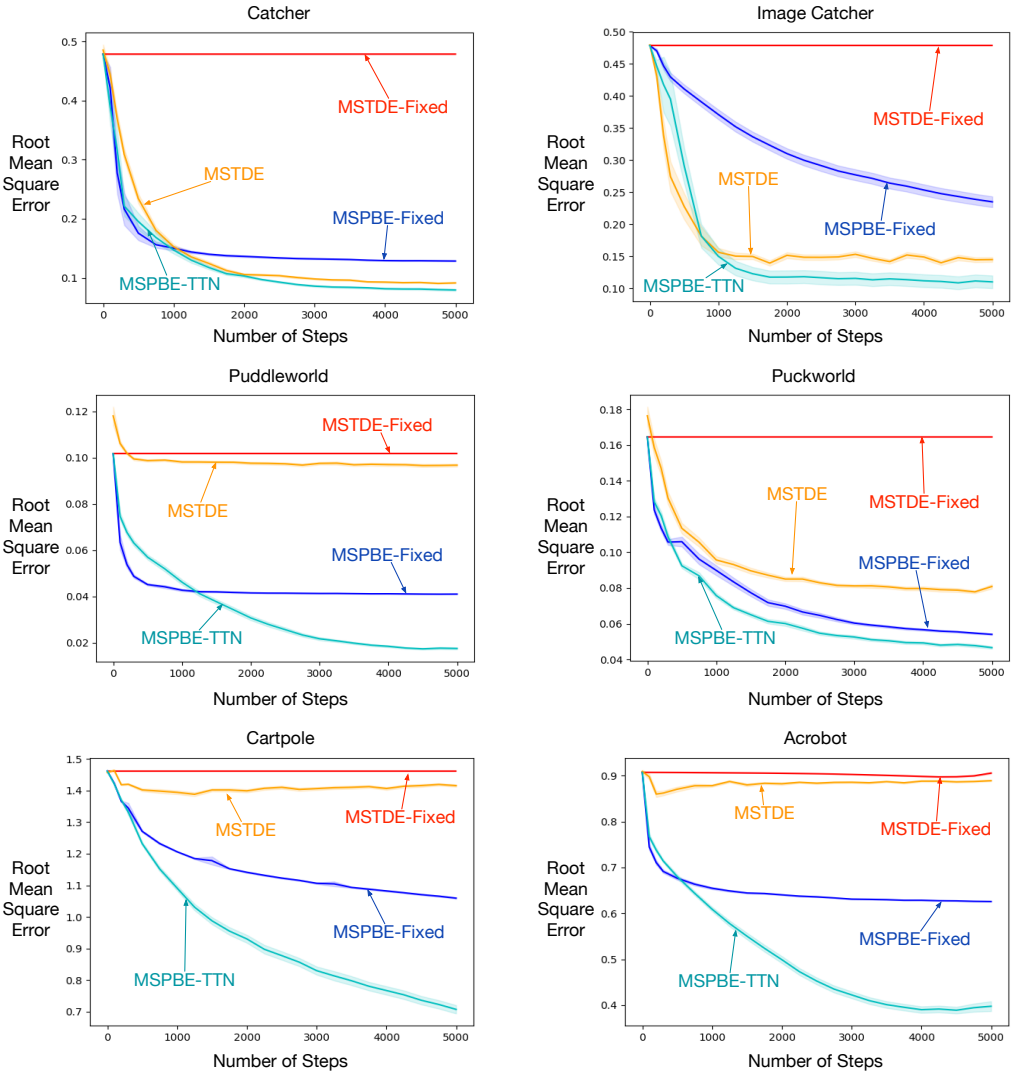


Figure 4.5: Learning curves under four different combinations corresponding to using the MSPBE or MSTDE for value-learning and fixed vs. learned features. We see that utilizing the MSPBE to learn the value function is superior to using the MSTDE (comparison of MSPBE-TTN and MSTDE curves). Also, the MSTDE can indeed be helpful for driving feature-learning (comparison of MSPBE-TTN and MSPBE-Fixed curves).

4.5 Surrogate Objectives

The TTN architecture allows us to flexibly specify the objective for driving the representation-learning aspect. In the previous experiments, we focused on the MSTDE since it was a simple objective which is directly related to our ultimate goal, predicting the value function. It is certainly possible to choose other losses instead and, in fact, finding suitable objectives for this purpose has been the subject of much research under the branches of auxiliary tasks or self-supervised learning. For example, the UNREAL architecture [17] proposes to make the agent learn to control pixels or the activations of hidden units as an auxiliary task to help learn useful representations.

Here, we investigate some simple alternatives to the MSTDE as a surrogate objective: predicting the next state, the next reward, or using the semi-gradient version of the MSTDE (treating the next state's value as fixed). The first two objectives are theoretically-sound since we are performing gradient descent on a fixed loss function (the targets are fixed for each step). On the other hand, the semi-gradient MSTDE is a common choice even though it does not (currently) have any theoretical guarantees with neural networks.

For these experiments, we use LSTD for the linear part for each of the objectives since it performed best. As opposed to the other experiments, we also sweep over the learning rate for the feature-learning part to accommodate the different objectives.

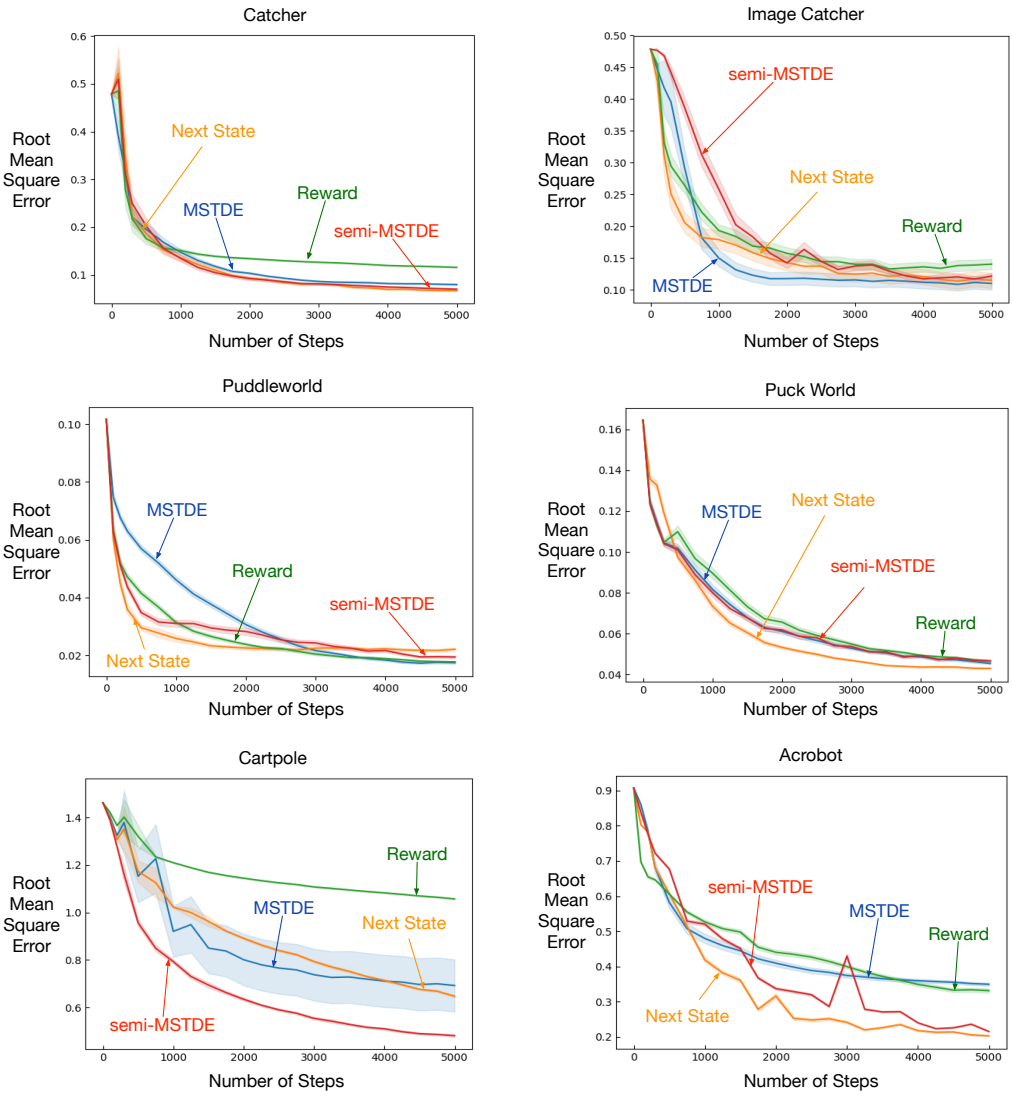


Figure 4.6: Learning curves under various surrogate losses. We find that there is no clear pattern as to which surrogate loss is superior, with the best one varying depending on the environment.

From figure 4.6 we find that the performance of the different objectives varied greatly across domains. In some environments such as Puck World, these loss functions all performed similarly while in others like Cartpole, there was greater variance between them. Overall, these results seem to indicate that the correct surrogate objective is task-dependent and that some experimentation would be required to find the best. It is notable that none of the surrogate objectives completely failed on any of the environments though, suggesting

these are all reasonable choices.

Further research into surrogate objectives may lead us to more principled approaches into choosing them. For example, one line of research is exploring using geometrical properties of value functions to produce suitable feature-learning objectives [9].

Chapter 5

Control

In this chapter, we present some experiments utilizing TTNs in the control setting. As baseline algorithms, we include Deep Q-Networks (DQN) [26] and least-squares DQN (LS-DQN) [21]. Inspired by fitted Q-iteration [12, 29], DQN adapts the classic Q-learning algorithm for use with neural networks by introducing two main additions to stabilize training: experience replay [22] and target networks.

To adapt TTNs to the control setting, we also make several modifications. We introduce a replay buffer to mitigate the difficulties of learning online with neural networks. This buffer is used for both feature and value-learning, though in different ways. For feature-learning, at every step, a mini-batch from the replay buffer is sampled and a gradient step is performed on it, minimizing the semi-gradient MSTDE. For value-learning, we directly use fitted Q-iteration (FQI) [12] for linear function approximation, treating the whole replay buffer as a batch of data. In more detail, the Q-learning targets, $y = r + \gamma \max_{a'} Q(s', a')$ are computed for every transition in the replay buffer. Then, to obtain the new set of value weights, we solve a linear regression problem where y is the response and $\boldsymbol{x}_{\boldsymbol{\theta}}(s)$ are the predictors. This solution can efficiently be found with standard solvers in $O(d^2 n)$ time, where d is the size of the last hidden layer of the network and n is the length of the replay buffer. Since this is a relatively expensive operation, the weights are only recomputed every m steps, reducing the amortized per-step cost to $O(d^2 n/m)$.

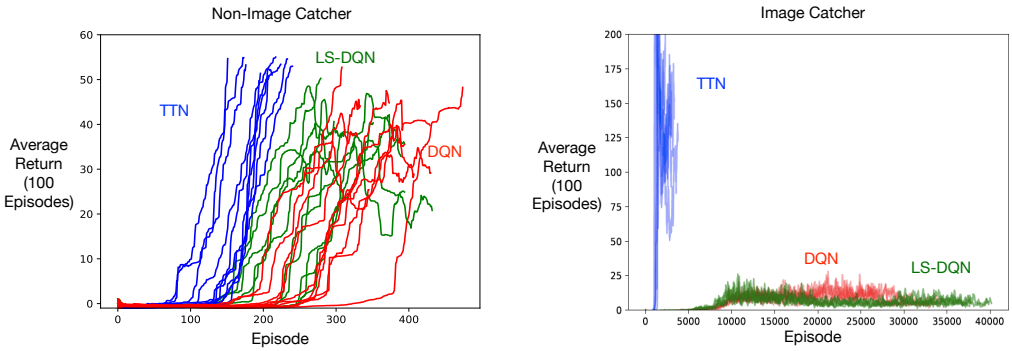
Unlike the original FQI algorithm designed for a fixed batch of data, we

consider the online setting, with a sliding window of experience being accumulated in the replay buffer. Due to this difference, we incorporate a regularization term in the FQI objective to encourage the linear regression solution to remain close to the current weights, just as in LS-DQN [21]. This can help prevent the parameters from changing wildly across consecutive FQI solves when the data in the buffer has changed.

The LS-DQN variant augments the standard DQN by periodically updating the final layer's set of weights towards the solution obtained by linear FQI, using the final hidden layer as the features. Once again, the primary difference between LS-DQN and TTN is that LS-DQN trains the representation jointly with the value estimates, while in TTN, these two processes are separated.

We evaluate the algorithms on Catcher, the nonimage and image versions. For nonimage (image) Catcher, we run the algorithms for 200 thousand (10 million) steps. The replay buffer was set to have a maximum size of 20 thousand (200 thousand) and initialized with 5000 (50000) steps using a random policy for non-image (image) Catcher. The neural network architectures were similar for all three algorithms: for nonimage catcher, the network consisted of three hidden layers of 128 units with ReLU activations. For image catcher, the chosen architecture was similar to the original DQN's [26], with two convolutional layers followed by a fully-connected layer of 256 units. To make comparisons more fair, the frequency for which the target network was updated for DQN/LS-DQN and TTN performed FQI solves were set to the same value. This ensures that the Q-learning targets were updated the same number of times for all the algorithms. These updates were done every 1000 (10000) steps for non-image (image) Catcher.

Concerning the other hyperparameters, on nonimage Catcher, we do a sweep over α_{slow} and λ_{reg}, the regularization parameter, for TTN and sweep over the learning rate and the number of steps over which ϵ is annealed for the DQN variants. Minibatches of size 32 were sampled from the replay buffer at every step to do updates. ϵ-greedy policies are used, for TTN, ϵ is fixed (for simplicity) while for DQN/LS-DQN it is annealed over time, as described originally. The final value of ϵ for all algorithms was set to 0.01 (0.1) for the



(a) (b)

Figure 5.1: Comparison of returns obtained by each algorithm on a) non-image Catcher and b) image Catcher.

non-image (image) versions. Each algorithm is run 10 times (5 times) for 200 thousand steps (10 million steps) on the non-image (image) Catcher.

From Fig. 5.1, we find that TTNs show competitive performance compared to both DQN variants. TTNs display a marked improvement early in training. While DQN catches close to zero apples for a large number of steps before suddenly improving, the length of this initial "warmup" phase is minimal for TTN. Our hypothesis is that this advantage is brought by the use of a least-squares method, fitted Q-iteration, for the value-learning process. Q-learning, the core learning mechanism in DQN, is a stochastic approximation algorithm which finds a good solution through the iterative process of taking a large number of small noisy steps. Hence, due to the small steps and noisy updates, convergence could be slow. On the other hand, fitted Q-iteration with linear function approximation is directly able to compute a closed-form solution on a whole batch of data. This can integrate any new information the agent observes quickly, without having to rely on noisy gradient steps to readjust the current set of weights. The experiments of Levine et al. support this hypothesis as they find that larger batch sizes with DQN are advantageous for learning [21]. Interestingly, LS-DQN does not seem to perform better than the vanilla DQN. This can be due to the fact that the FQI updates in LS-DQN make large changes to the weights in the final layer. Since there is only one set of weights for the last layer in LS-DQN, this may destabilize the learning process for the earlier layers.

On image Catcher, we can also observe that all three algorithms seem to be prone to forgetting; the average return increases at first but then decreases significantly after some amount of time. TTN is particularly affected by this, as the average return reaches almost 500 before dipping back down to around 100. A plausible explanation is that, when the policy is close to optimal, the replay buffer gets filled exclusively with the data from certain, near optimal, trajectories. Since the training data no longer covers the state space effectively, the agent forgets the optimal policy in many states. Thus, if the agent happens to land in one of these rare states, it would not be able to perform well. This can have a compounding effect as the agent then has to learn the action-values in those states again which themselves bootstrap off incorrect values from other rare state-action pairs in Q-learning. Thus, the agent's performance can suffer significantly. By looking at the number of steps elapsed in the upwards spike in TTN's average return, we find that the average return is greater than 200 for over a million steps, much larger than the replay buffer's capacity of 200 thousand transitions. This lends credence to the previous forgetting hypothesis, as the buffer could be completely refilled by transitions from a near-optimal policy. Note that, despite the drop, TTN's average return still remains higher than the DQN algorithms' at any point during training, and slowly increases over time thereafter.

In conclusion, we find that TTNs can perform well in control in comparison to end-to-end methods such as DQN and LS-DQN. By utilizing FQI on entire replay buffer, TTNs can learn more quickly, especially at the beginning of training.

Chapter 6

Conclusion and Future Directions

6.1 Contributions

In this thesis, we investigated the Two-Timescale Network (TTN), a new architecture for learning value functions which are nonlinear functions of the state. The proposed architecture separates the feature and value-learning processes, a departure from the standard deep RL approach of learning both parts jointly. This separation enable the use of many classic linear policy evaluation algorithms which do not have any sound extensions to nonlinear function approximation. We also presented a theorem showing that this training scheme is guaranteed to converge under standard assumptions for temporal-difference algorithms.

We performed a suite of experiments to investigate various design choices related to TTNs. We found that the most effective linear algorithms for value-learning were the least-squares methods, whose main drawback in terms of computational costs are offset by choosing a relatively small number of features. Additionally, eligibility traces were found to be an effective choice for many of the linear algorithms although it was not beneficial when used naively with nonlinear TD (a theoretically-unsound algorithm). The choice of surrogate loss was found to be environment-dependent and no surrogate loss universally outperformed the others. Finally, experiments in the control setting with TTN combined with fitted Q-iteration showed that TTN's can greatly outperform DQN.

6.2 Future Directions

The core idea of separating feature-learning and value-learning is general and there are many aspects to be examined. We outline a couple of possible research direction that may warrant further investigation.

Off-policy Learning and Variance Reduction

When the main objective is difficult to optimize, it may be easier to separate the learning problem into two pieces: learning a representation and then tackling the main task. By choosing a simpler surrogate objective to propel representation-learning, it can be easier to obtain reasonable (though perhaps not optimal) features. These features can then be leveraged with more powerful optimizers developed for linear function approximation to obtain predictions for the main task. Overall, by reducing the noise in the optimization process, we can expect the separation approach to lead to faster learning. Since off-policy learning typically leads to higher variance gradients, it seems like a suitable setting to examine to magnify the effect of any variance reduction. Investigations into this hypothesis could be an interesting direction for future work. Extending this idea further, it may be even more effective to decompose an agent's learned representation into separate modules, each with its own surrogate objective to optimize. This may potentially reduce the variance even further by decomposing the entire (difficult) optimization problem into subproblems that are less noisy and easier to optimize.

Surrogate Objectives and Representation-learning

In this work, we did not find any conclusive evidence about which surrogate objective was best. Since we only tried simple surrogate objectives, it is likely that these are not robust to different environments. It would be interesting to answer questions such as:

- What properties do the learned representations have for each surrogate objective?
- What kind of representation do we want to learn?
- How can we design a surrogate objective to ensure that we learn desired

representations?

Understanding Control Experiments

We found that TTN utilizing linear fitted Q-iteration was able to soundly outperform DQN in the Catcher environment. While promising, it would be good to confirm that TTNs are able to work effectively in a larger range of environments. Additionally, looking to understand more deeply why TTN-FQI gives much better performance, particularly during the early training phase, would be of interest. A plausible hypothesis would be that using FQI to directly solve for a good set of weights is much more effective than having to a larger number of stochastic gradient steps as in DQN. It has been observed previously that typical learning curves for DQN are flat for some large number of steps at the beginning before suddenly jumping up. This may be due to the fact that it takes a long time for the Q-values to be reasonably accurate starting from a random initialization since each gradient step only makes small noisy changes to the parameters. On the other hand, FQI can find the best set of weights in a single solve, limiting the length of the 'warmup' phase.

References

[1] Adam Adam and Martha White. "Investigating practical linear temporal difference learning." In: *Proceedings of the 2016 International Conference on Autonomous Agents & Multiagent Systems*. International Foundation for Autonomous Agents and Multiagent Systems. 2016, pp. 494–502. 18

[2] Leemon Baird. "Residual algorithms: Reinforcement learning with function approximation." In: *Machine Learning Proceedings 1995*. Elsevier, 1995, pp. 30–37. 10

[3] Vivek S Borkar. *Stochastic approximation: a dynamical systems viewpoint*. Vol. 48. Springer, 2009. 15

[4] Léon Bottou, Frank E Curtis, and Jorge Nocedal. "Optimization methods for large-scale machine learning." In: *Siam Review* 60.2 (2018), pp. 223–311. 18

[5] Justin A Boyan and Andrew W Moore. "Generalization in reinforcement learning: Safely approximating the value function." In: *Advances in neural information processing systems*. 1995, pp. 369–376. 21

[6] Steven J Bradtke and Andrew G Barto. "Linear least-squares algorithms for temporal difference learning." In: *Machine learning* 22.1-3 (1996), pp. 33–57. 2, 12, 15, 26

[7] Wesley Chung, Somjit Nath, Ajin Joseph, and Martha White. "Two-timescale networks for nonlinear value function approximation." In: *Proceedings of the International Conference on Learning Representations (ICLR)*. 2019. 17

[8] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. "Fast and accurate deep network learning by exponential linear units (elus)." In: *arXiv preprint arXiv:1511.07289* (2015). 18

[9] Robert Dadashi, Adrien Ali Taiga, Nicolas Le Roux, Dale Schuurmans, and Marc G Bellemare. "The Value Function Polytope in Reinforcement Learning." In: *arXiv preprint arXiv:1901.11524* (2019). 36

[10] Christoph Dann, Gerhard Neumann, and Jan Peters. "Policy evaluation with temporal differences: A survey and comparison." In: *The Journal of Machine Learning Research* 15.1 (2014), pp. 809–883. 10, 12

[11] Dotan Di Castro and Shie Mannor. "Adaptive bases for reinforcement learning." In: *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer. 2010, pp. 312–327. 3, 18, 24

[12] Damien Ernst, Pierre Geurts, and Louis Wehenkel. "Tree-based batch mode reinforcement learning." In: *Journal of Machine Learning Research* 6.Apr (2005), pp. 503–556. 37

[13] Benjamin Eysenbach, Shixiang Gu, Julian Ibarz, and Sergey Levine. "Leave no trace: Learning to reset for safe and autonomous reinforcement learning." In: *arXiv preprint arXiv:1711.06782* (2017). 1

[14] Rong Ge, Furong Huang, Chi Jin, and Yang Yuan. "Escaping from saddle points—online stochastic gradient for tensor decomposition." In: *Conference on Learning Theory*. 2015, pp. 797–842. 17

[15] Xavier Glorot and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks." In: *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. 2010, pp. 249–256. 21

[16] Matteo Hessel, Joseph Modayil, Hado Van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. "Rainbow: Combining improvements in deep reinforcement learning." In: *Thirty-Second AAAI Conference on Artificial Intelligence*. 2018. 18

[17] Max Jaderberg, Volodymyr Mnih, Wojciech Marian Czarnecki, Tom Schaul, Joel Z Leibo, David Silver, and Koray Kavukcuoglu. "Reinforcement learning with unsupervised auxiliary tasks." In: *arXiv preprint arXiv:1611.05397* (2016). 34

[18] Chi Jin, Rong Ge, Praneeth Netrapalli, Sham M Kakade, and Michael I Jordan. "How to escape saddle points efficiently." In: *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org. 2017, pp. 1724–1732. 17

[19] Michail G Lagoudakis and Ronald Parr. "Least-squares policy iteration." In: *Journal of machine learning research* 4.Dec (2003), pp. 1107–1149. 26

[20] David A Levin and Yuval Peres. *Markov chains and mixing times*. Vol. 107. American Mathematical Soc., 2017. 16

[21] Nir Levine, Tom Zahavy, Daniel J Mankowitz, Aviv Tamar, and Shie Mannor. "Shallow updates for deep reinforcement learning." In: *Advances in Neural Information Processing Systems*. 2017, pp. 3135–3145. 3, 23, 37–39

[22] Long-Ji Lin. "Self-improving reactive agents based on reinforcement learning, planning and teaching." In: *Machine learning* 8.3-4 (1992), pp. 293–321. 37

[23] Hamid R Maei, Csaba Szepesvári, Shalabh Bhatnagar, Doina Precup, David Silver, and Richard S Sutton. "Convergent temporal-difference learning with arbitrary smooth function approximation." In: *Advances in Neural Information Processing Systems*. 2009, pp. 1204–1212. 24

[24] Hamid Reza Maei. "Gradient temporal-difference learning algorithms." In: (2011). 11

[25] Ishai Menache, Shie Mannor, and Nahum Shimkin. "Basis function adaptation in temporal difference reinforcement learning." In: *Annals of Operations Research* 134.1 (2005), pp. 215–238. 3

[26] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. "Playing atari with deep reinforcement learning." In: *arXiv preprint arXiv:1312.5602* (2013). 2, 18, 37, 38

[27] KJ Prabuchandran, Shalabh Bhatnagar, and Vivek S Borkar. "Actor-Critic Algorithms with Online Feature Adaptation." In: *ACM Transactions on Modeling and Computer Simulation (TOMACS)* 26.4 (2016), p. 24. 3

[28] Sashank J Reddi, Satyen Kale, and Sanjiv Kumar. "On the convergence of adam and beyond." In: (2018). 21

[29] Martin Riedmiller. "Neural fitted Q iteration–first experiences with a data efficient neural reinforcement learning method." In: *European Conference on Machine Learning*. Springer. 2005, pp. 317–328. 37

[30] Tom Schaul, Daniel Horgan, Karol Gregor, and David Silver. "Universal value function approximators." In: *International Conference on Machine Learning*. 2015, pp. 1312–1320. 1

[31] Ralf Schoknecht and Artur Merke. "TD (0) converges provably faster than the residual gradient algorithm." In: *Proceedings of the 20th International Conference on Machine Learning (ICML-03)*. 2003, pp. 680–687. 33

[32] Harm Seijen and Rich Sutton. "True online TD (lambda)." In: *International Conference on Machine Learning*. 2014, pp. 692–700. 12

[33] Harm van Seijen, A Rupam Mahmood, Patrick M Pilarski, and Richard S Sutton. "An Empirical Evaluation of True Online TD ({\lambda})." In: *arXiv preprint arXiv:1507.00353* (2015). 12

[34] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. "Mastering the game of go without human knowledge." In: *Nature* 550.7676 (2017), p. 354. 2

[35] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. 2018. 7, 9, 10, 33

[36] Richard S Sutton, A Rupam Mahmood, and Martha White. "An emphatic approach to the problem of off-policy temporal-difference learning." In: *The Journal of Machine Learning Research* 17.1 (2016), pp. 2603–2631. 2, 11

[37] Richard S Sutton, Hamid Reza Maei, Doina Precup, Shalabh Bhatnagar, David Silver, Csaba Szepesvári, and Eric Wiewiora. "Fast gradient-descent methods for temporal-difference learning with linear function approximation." In: *Proceedings of the 26th Annual International Conference on Machine Learning*. ACM. 2009, pp. 993–1000. 2, 15, 18

[38] Richard S Sutton, Joseph Modayil, Michael Delp, Thomas Degris, Patrick M Pilarski, Adam White, and Doina Precup. "Horde: A scalable real-time architecture for learning knowledge from unsupervised sensorimotor interaction." In: *The 10th International Conference on Autonomous Agents and Multiagent Systems-Volume 2*. International Foundation for Autonomous Agents and Multiagent Systems. 2011, pp. 761–768. 1, 7

[39] Norman Tasfi. *PyGame Learning Environment*. `https://github.com/ntasfi/PyGame-Learning-Environment`. 2016. 21

[40] John N Tsitsiklis and Benjamin Van Roy. "Analysis of temporal-diffference learning with function approximation." In: *Advances in neural information processing systems*. 1997, pp. 1075–1081. 2, 15, 23

[41] Harm Vanseijen and Rich Sutton. "A deeper look at planning as learning from replay." In: *International conference on machine learning*. 2015, pp. 2314–2322. 12

[42] Martha White. "Unifying task specification in reinforcement learning." In: *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org. 2017, pp. 3742–3750. 6

[43] Huizhen Yu and Dimitri P Bertsekas. "Basis function adaptation methods for cost approximation in MDP." In: *2009 IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning*. IEEE. 2009, pp. 74–81. 3