Decomposing the Bellman Equation in Reinforcement Learning

Joshua Romoff

A thesis submitted to McGill University in partial fulfillment of the requirements of the degree of Doctor of Philosophy

School of Computer Science, McGill University, Montreal

April 2021

Contents

Abstract

Artificial intelligence (AI) research is centered around designing scalable learning systems that can solve complex tasks in an efficient manner. The main tool that we analyze in this thesis is deep reinforcement learning (deep RL), which provides a general learning framework for AI systems to solve sequential decision making problems. Deep RL, however, can often require millions of samples to attain near-human performance and is often highly sensitive to both noise in the environment and hyperparameter choices. In this thesis, we argue that these shortcomings can be addressed by decomposing the value estimation problem into separate components that can be learned in parallel. To this end, we first consider decomposing the reward function into many factors and learning a separate value function for each separate component. Since each individual value function typically only depends on a subset of all features, the value estimate can be approximated more easily by a low-dimensional representation, enabling faster learning. Next, we suggest learning an estimator for the expected reward in order to train the value function. We demonstrate that this simple decomposition improves performance under stochastic rewards in many deep RL applications. Furthermore, we extend this approach to value functions by using shorter horizon values as a training signal for longer horizons. The separation of value functions into these shorter horizon components has useful properties in scalability and performance. We then unify these decompositions under one unifying theorem, which reveals the conditions under which we may expect the decompositions to differ from traditional approaches. Crucially, we discover that we can break the equivalence with non-decomposed approaches by using a different learning rate for each decomposed value function. Thus, our final work analyzes the use of Jacobi preconditioning in TD learning, which prescribes a principled approach for a per parameter learning rate. We find that once the global learning rate has been tuned, Jacobi preconditioning is competitive with state of the art adaptive optimizers.

Abrégé

La recherche sur l'intelligence artificielle (IA) est centrée sur la conception de systèmes d'apprentissage capables de résoudre des tâches complexes de manière efficace. Le principal outil que nous analysons dans cette thèse est l'apprentissage par renforcement profond (deep RL), qui fournit un cadre d'apprentissage général aux systèmes d'IA pour résoudre des problèmes de prise de décision séquentielle. Cependant, Deep RL peut souvent nécessiter des millions d'éxamples pour atteindre des performances quasi humaines et est souvent très sensible au bruit dans l'environnement et aux choix d'hyperparamètres. Dans cette thèse, nous soutenons que ces faults peuvent être surmontées en décomposant le problème d'estimation de valeur en composantes distinctes qui peuvent être apprises en parallèle. À cette fin, nous envisageons d'abord de décomposer la fonction de récompense en plusieurs facteurs et d'apprendre une fonction de valeur distincte pour chaque composant. Étant donné que chaque fonction de valeur individuelle ne dépend généralement que d'un sous-ensemble de toutes les caractéristiques, l'estimation de la valeur peut être approchée plus facilement par une représentation de faible dimension, permettant un apprentissage plus rapide. Ensuite, nous suggérons d'apprendre un estimateur de la récompense attendue afin d'entraîner la fonction de valeur. Nous démontrons que cette simple décomposition améliore les performances sous récompenses stochastiques dans de nombreuses applications deep RL. De plus, nous étendons cette approche aux fonctions de valeur en utilisant des valeurs d'horizon plus courtes comme signal d'apprentissage pour des horizons plus longs. La séparation des fonctions de valeur en ces composants d'horizon plus courts a des propriétés utiles en termes d'évolutivité et de performances. Nous unifions ensuite ces décompositions sous un théorème unificateur, qui révèle les conditions dans lesquelles nous pouvons nous attendre à ce que les décompositions diffèrent des approches traditionnelles. Fondamentalement, nous découvrons que nous pouvons rompre l'équivalence avec des approches non décomposées en utilisant un taux d'apprentissage différent pour chaque fonction de valeur décomposée. Ainsi, notre travail final analyse l'utilisation du préconditionnement Jacobi dans l'apprentissage TD, qui prescrit une approche de principe pour un taux d'apprentissage par paramètre. Nous constatons qu'une fois que le taux d'apprentissage global a été réglé, le préconditionnement Jacobi est compétitif avec les optimiseurs adaptatifs de l'état de l'art.

Acknowledgements

I want to start by thanking all of my wonderful friends, family, and collaborators, whose support helped me get to where I am today.

Thank you Meghan, for being the most supportive partner one could hope for and for proofreading a draft of every paper I have ever written (including this thesis).

Thank you Joelle, for being an incredibly insightful and supportive supervisor over the years. Most of all, I'd like to thank you for giving me the opportunity of a lifetime and for guiding me along this path.

Thank you Peter, without your help most of the work in this thesis would have not been possible. Thank you Ahmed, you're a mathematical guru who came to my rescue many times. Thank you Mido, you're positive outlook kept me going.

Finally, I'd like to thank my PhD committee (proposal and defense) for taking the time to provide me with useful feedback: Pierre-Luc Bacon, Marc G. Bellemare, David Rolnick, Jérôme Waldispühl, and Adam White.

Contribution to Original Knowledge

This thesis contributes to the field of deep reinforcement learning by proposing several techniques to effectively solve the value estimation problem. Specifically, we propose to:

1. Decompose the reward function into several components based off the underlying state space, this allows for:
    - Regularized training of value functions, that can be trained independently using a reduced state space.
2. Use an estimate of the reward function as a replacement for its empirically sampled counterpart, which provides:
    - Variance reduction and improved sample efficiency in certain tabular and deep RL settings, particularly with stochastic reward functions.
3. Decompose the value function based off of the discount factor, leading to:
    - A Bellman-like equation to learn the differences between value functions with different discount factors.
    - A principled approach for setting certain bias-variance hyperparameters.
4. Unify TD based decompositions via the Bellman decomposition theorem, which identifies potential areas for improvement, including:
    - Using a different n-step λ-return for each decomposed value function.
    - Using a different learning rate for each decomposed value function.
5. Integrate Jacobi preconditioning with temporal difference methods leading to:
    - The extension of Jacobi matrix splitting to the multi-step TD regime.
    - An adaptive optimizer that tunes the learning rate on a per parameter basis.

Contribution of Authors

- Chapters 1 and 2 are written specifically for this thesis.
- Chapter 3 is based on a conference paper (van Seijen et al., 2017) that was submitted during my time as an intern at Microsoft Research Montreal. My contributions to the work were in aiding in the development of the algorithm, implementing and running the deep RL algorithms for the toy experiment in section 3.4.1, and implementing and running the A3C baselines in section 3.4.2. The conception of the algorithm was by Harm van Seijen. The tabular algorithms, including the main results on Ms. Pac-Man were conducted by Mehdi Fatemi, Romain Laroche, and Harm van Seijen. Most of the paper was written by Harm van Seijen. Finally, the feature extraction was done by Jeffery Tsang and Tavian Barnes. Follow up work in Laroche et al. (2017), further analyzed the different objective functions analyzed in section 3.2.
- Chapter 4 is based on a conference paper (Romoff et al., 2018a) that evolved from a workshop paper (Romoff et al., 2018b). Interestingly, we had already derived the Bellman-like equation for TD(Δ) (presented in chapter 5) and it was only after many failed experiments that we decided to try using reward estimation (its base case). My contribution to the work was in developing the main algorithm, developping the theory in section 4.3, and in conducting the experiments. Peter Henderson co-wrote the paper, helping with the overall organization of the paper and the main experimental decisions. Alexandre Piché and Viçent Francois-Lavet aided in the development of the main ideas and writing the paper. Joelle Pineau supervised the project by providing useful feedback, helping with the writing, and crucially giving us that extra bit of encouragement when nothing seemed to be working.
- Chapter 5 is based on a conference paper (Romoff et al., 2019). My contributions were developing the main algorithm, proving the equivalence theorem (theorem 5.3.1), and in coding and running the deep RL experiments. Yann Ollivier and I, independently derived the main Bellman equation. Conveniently, we were both working at Facebook AI Research at the time and enthusiastically decided to combine forces and work

together. The paper would not have been possible without Peter Henderson's various contributions including: co-writing the paper, in deriving most of theorem 5.3.4, and in running the tabular experiments (not included in this manuscript but available in the main paper). Ahmed Touati greatly helped in organizing and verifying all of the math in the paper, and also derived theorem 5.3.2 (which allows for $\lambda > 1$). Emma Brunskill derived thereom 5.3.3 from (Kearns and Singh, 2000), and helped with the writing. Finally, Joelle Pineau and Yann Ollivier were the main supervisors for the project, providing constant feedback throughout the more than one year process, and both contributed to the writing of the paper.

- Chapter 6 is written specifically for this thesis.

- Chapter 7 is based on work presented at two workshops at ICML (Romoff et al., 2020): *theoretical foundations of reinforcement learning workshop* and *beyond first order optimization workshop*. My contribution was in developing the algorithm and theory of the chapter, as well as writing the code for the optimizer and the experiments. Peter Henderson contributed immensely in many one on one discussions (for over a year), helped with the implementation, ran the regression analysis (not included in this manuscript but is included in the cited version), and also greatly helped with the writing of the paper. David Kanaa was involved in several one on one discussions and helped develop and organize the theory sections of the paper. Emmanuel Bengio ran the deep RL experiments and helped with the writing of the paper. Ahmed Touati helped develop the theory in the early goings of the project, and helped write the paper. Pierre-Luc Bacon was instrumental in pushing the project in the direction of stochastic optimization, without him the project would have forever been stalled. Finally, Joelle supervised the project, throughout all of its ups and (many) downs, and helped with the writing.

- Throughout my PhD studies I have also been fortunate enough to collaborate with many amazing people on several other works that are not included in this thesis (van Seijen et al., 2016; Laroche et al., 2017; Henderson et al., 2018; Touati et al., 2018; Das et al., 2019; Assran et al., 2019; Henderson et al., 2020).

Chapter 1

Introduction

Artificial intelligence (AI) research is centered around designing scalable learning systems that can solve complex tasks in an efficient manner. Ideally, these systems should be able to interact with the world around them and generalize past experiences to novel, yet similar, scenarios. The types of tasks that we would like to be solved by machines are varied; ranging from classification tasks to sequential decision-making problems. While the former is fairly well understood, the latter, specifically creating a general purpose sequential decision making agent, remains an open problem.

The main tool that we analyze in this thesis is the deep reinforcement learning (deep RL) framework, which combines the generalization power of deep learning (DL) (Goodfellow et al., 2016) and the sequential decision making framework of reinforcement learning (RL) (Sutton and Barto, 2018). Deep reinforcement learning has endless amount of applications, which notably include both the Healthcare (Yu et al., 2019) and Finance (Jiang et al., 2017) sectors. Recently, deep RL has led to tremendous research breakthroughs; including but not limited to, playing Atari games at a near human level (Mnih et al., 2015) and defeating the world's best GO player (Silver et al., 2016a). However, deep RL can often be:

- Brittle: high sensitivity to noise and/or hyperparameters.
- Sample inefficient: needing millions of samples to attain near-human level performance.

Solving these deficiencies remains the main bottleneck preventing deep RL from entering into mainstream products. To this end, in this thesis, we propose new methods to mitigate the brittleness and sample inefficiency of deep RL systems by decomposing the problem into simple components that can be solved in parallel.

1.1 Reinforcement Learning

The notion of training through reinforcement is an old concept that began in behavioural psychology. In the experiments of Thorndike (1898), cats were trained to escape mazes by placing a visible morsel of food at the end of a puzzle. Similarly, RL in an AI context, aims to train systems to perform tasks by providing positive or negative feedback. However, unlike in behavioural psychology, reinforcement learning is rooted in the theory of Markov decision processes (MDP) (Bellman, 1957), which allows for formal mathematical formulation and analysis.

In an MDP, an agent interacts with a specific environment with the goal of learning a *policy* to maximize the discounted sum of a predefined *reward* signal. The discount factor plays the role of giving more importance to rewards that are closer in time to the agent. At any point in time, the agent is in a given *state* of the world and has to take an *action*. Upon taking an action, the agent transitions to another state and receives a *reward* based off of a predefined *transition probability distribution* and *reward function* from the environment.

RL agents aim to solve MDPs under the constraint that the agent does not have access to the underlying transition probability distribution or reward function. In other words, before interacting with the environment, the agent does not know which state and action pairs are good, or even how to go from one state of the world to another. All of this knowledge needs to be acquired through interacting with the environment. To use RL, practitioners need to ensure that their task fits into the MDP framework. One key assumption of the MDP framework, is the *Markov property*, which assumes that future states are not dependant on past states given the current state of the environment.

The two main types of RL methods that we will cover in this thesis are *actor-critic* methods (Sutton et al., 2000) and *value-based* methods (Sutton, 1988; Watkins, 1989; Rummery and Niranjan, 1994). Both of which rely on estimating the value function: the expected sum of discounted rewards under the current policy. The agent then uses the value function directly to take actions in value-based methods, or uses it to improve a policy in actor-critic methods.

The most commonly used method for training value functions is *temporal-difference* (TD) learning (Sutton, 1988), which is rooted in the *Bellman equation* (Bellman, 1957). The Bellman equation provides a recursive formulation for the value function in terms of the value at the next time-step. Specifically, instead of using the sum of discounted rewards sampled from a particular state in the environment, TD learning uses the immediate reward and the discounted value at the next time-step as a training target. TD learning will be central to this thesis, with a particular focus on its use in deep reinforcement learning, which we motivate and describe in the next section.

1.2 Deep Reinforcement Learning

In recent years, the combination of deep learning and reinforcement learning has been used with great success in achieving super-human performance in complicated games (Mnih et al., 2015; Silver et al., 2016a; Vinyals et al., 2019). However, it may not be immediately obvious to the reader what the *deep* part contributes to in deep reinforcement learning. We shed some light on this question with the following example.

Consider an agent tasked with solving *Pong*, which is a tennis like video game from the *Atari 2600* console, also available as a training simulator for AI systems via the *arcade learning environment* (ALE) (Bellemare et al., 2013). The goal of the game is to prevent the ball from getting past your own paddle while trying to hit the ball past the enemies paddle on the other side of the screen. At a conceptual level, the problem is fairly simple, move the paddle up if the ball appears to be going towards the top of the screen and vice versa. Complications arise when trying to formulate the states of the world in the RL framework. Since the game is viewed on a screen with many pixels, the number of states is enormous. Even in the simplified black and white version, where colours have been removed, the combination of pixels (being on or off) with a resolution of $h \times w$ would result in $2^{h \times w}$ number of states. Accounting for the ball's velocity, to ensure the Markov property, would require a short history of images, further increasing the theoretical[1] number of states.

The difficulty with a large number of states is that for an agent to learn how to act optimally in a particular state, a certain amount of samples needs to be obtained from that state. Therefore, if the total number of states grows, so does the overall amount of samples needed. To alleviate this problem, termed *sample efficiency*, some form of *perceptual generalization* needs to occur, where samples from a similar state are also incorporated.

One solution is for a programmer to design a *feature extractor* that extracts the required information from the screen at each time-step; e.g, the position of both paddles as well as the position and velocity of the ball. Such a technique would indeed reduce the state space down significantly, perhaps to the point where a simple linear mapping, termed *linear function approximator*, can be learned. Unfortunately, such an approach may not be general. If presented with a similar game, say *Ms. Pacman* (also from the Atari 2600 console), the programmer would have to reprogram a feature extractor for the new game. While there are feature extraction methods that have been shown to perform well across many tasks (Liang et al., 2015), in recent years, Deep RL solutions have greatly outperformed these algorithms.

Deep RL aims to solve the problem of feature extraction by providing an end to end solution for learning both the agent's policy and the feature extractor itself, through the

[1]Not all of these states would be visitable, since certain combinations of pixels never occur in the game.

use of *non-linear function approximators*. For image based tasks, such as the video games described above, deep RL is used with powerful *neural network* architectures such as modern versions of the *convolutional neural network* (CNN) (LeCun et al., 1989). However, even with the addition of deep learning, deep RL can still be sample inefficient, typically needing hundreds of millions of interactions with the world to obtain near-human performance (Mnih et al., 2015). Moreover, deep RL architectures can be brittle, in the sense that they may fail to learn meaningful policies when presented with noise in the environment or when the practitioner fails to set certain hyperparameters properly. We further motivate the issues of sample efficiency and brittleness in the following sections.

1.3 Sample Efficiency

Sample efficiency is a metric used to evaluate reinforcement learning systems, which relates the amount of reward the agent is receiving to the number of interactions the agent has had with the environment. An agent that can achieve the same amount of reward as another with less experience is termed more sample efficient. The importance of sample efficiency comes from the assumption that samples from the environment are costly to obtain, where cost can be defined in terms of the amount of time spent, computational cost, or opportunity cost.

Consider training a robot in the real world where the amount of samples the robot is able to train with is bottle-necked by time itself. If the task is difficult, measured in terms of human capability, then the amount of time needed could be immense. In practice, however, we rarely train agents in the real world, but instead train them in simulators that can operate at many times faster than real-time (e.g, 100x in the arcade learning environment (ALE) (Bellemare et al., 2013)). Furthermore, we can also deploy several agents to collect samples in parallel, thus greatly reducing the overall time complexity. Both of these solutions are incredibly powerful and have led to the most advanced deep RL systems to date (Silver et al., 2016a; Vinyals et al., 2019).

Even if a simulator can be used, there are still several reasons for wanting a sample efficient system. First, as the models that deep RL agents use get larger and larger, the carbon footprint and overall energy requirement of these systems may grow as well (Henderson et al., 2020). The net savings, in terms of energy, could be enormous when considering the total number of applications for deep RL. Second, sample efficiency can be crucial in *non-stationary* environments, where agents are required to continuously adapt to changes to the world around them. For example, in financial stock trading the dynamics of the market are continuously changing (Lee et al., 2007); new players enter and exit the market at all times and opposing strategies are constantly changing. For both of these reasons, having a

system that can adapt with as few samples as possible is greatly desired.

1.4 Brittleness

In this section, we describe *brittleness*, one of the main obstacles to achieving a sample efficient system. We use the term brittleness to describe a system's weakness to external or internal forces. Specifically, there are two forms of brittleness that we address in this thesis: brittleness to noise from the external environment and brittleness to the agent's *hyperparameters*, i.e., the agent's internal settings that can be tuned. While brittleness to noise or hyperparameters is also an issue with standard RL, i.e., with linear function approximation, the problems are typically enhanced in deep RL due to the complexity of the applications and the overall complexity of the learning system. Both forms of brittleness can result in poor sample efficiency and even a failure to learn policies that are better than random (Henderson et al., 2018).

The main type of noise that we address in this thesis are noisy rewards. Noisy rewards can be caused by several sources, including noisy sensors (Everitt et al., 2017) and variable human feedback (Knox and Stone, 2012). For example, a robot that operates in the real world based off of raw sensor data may encounter periodic disruptions to the sensors. Moreover, an agent that is being trained off of human feedback, may receive noisy human feedback since humans are prone to making mistakes. In either case, an ideal learning system should be able to handle a moderate amount of noise in the reward signal and still successfully train an accurate value function.

Brittleness to hyperparameters can result in extensive tuning which can be time consuming for the programmer and computationally expensive to implement. Thus, part of the focus of this thesis is to design learning systems that are less sensitive to hyperparameter choices. The hyperparameters that we focus on in this thesis are the *learning rate* and the *discount factor*. The learning rate dictates how much an agent learns from each sample in the environment. Setting it too high can lead to *divergence*, whereas setting it too low can lead to poor sample efficiency (Sutton and Barto, 2018). On the other hand, the discount factor is defined as part of the MDP, thus, in classical MDP settings, is not meant to be treated as a tuneable hyperparameter. However, in many scenarios tuning the discount factor can lead to improved performance on the objective associated with the original discount factor (Bertsekas and Tsitsiklis, 1995).

1.5 Decomposing Reinforcement Learning

Motivated by the classical *divide-and-conquer* paradigm in computer science, we are interested in dividing the original RL problem into simpler components. The main goal of decomposing the reinforcement learning problem is to provide structure for the learning system to exploit. While not always the case, as we will see in chapter 3, this added structure may come at the expense of biasing the learning system towards potentially suboptimal solutions. Nevertheless, this added bias may drastically improve sample efficiency and thus may still be desired. While there are many ways to decompose the reinforcement learning problem, the two main types of decomposition that we consider in this thesis are *hierarchical* methods (Sutton et al., 1999; Bacon et al., 2017; Kulkarni et al., 2016), and *model-based* methods (Sutton, 1991; Feinberg et al., 2018; Kaiser et al., 2019).

Hierarchical methods typically divide the problem into several layers of blocks that each operate on top of one another (Sutton et al., 1999; Bacon et al., 2017; Kulkarni et al., 2016). These types of methods can be thought of as dividing up the problem into shorter and longer *time-scales* (Bacon, 2018). For example, in *options* (Sutton et al., 1999; Bacon et al., 2017), low-level agents operate on the raw action space of the problem, whereas high level agents select which low-level agent to use. We propose a simple hierarchical model in chapter 3, and explore decompositions based off of the time-scale in chapter 5.

Another relevant approach for decomposition are model-based techniques for *model-free* learning (Sutton, 1991; Feinberg et al., 2018). These methods train a model of the environments dynamics, and then use it to perform updates to the agents value function and/or policy (the model-free component). We can think of these methods as decomposing the learning problem into two components, learning the transition dynamics and learning the value function and/or policy. Such approaches typically offer variance reduction, since the agent does not need to sample experiences from a potentially stochastic environment. However, they come at the expense of added bias in the case where the model is imperfect (Sutton and Barto, 2018). We explore learning a simple model of the reward function in chapter 4 and of the transition dynamics in chapter 7.

1.6 Objectives and Outline

This thesis is organized as follows. Chapter 2 covers the necessary background for deep RL, with a particular focus on value estimation. In the proceeding chapters, we propose and analyze various methods that decompose the Bellman equation to efficiently solve the value estimation problem. We will mainly asses the performance of our approaches using the arcade

learning environment (Bellemare et al., 2013), which provides many complex pixel based tasks from the Atari 2600.

In many complicated domains, the reward can be characterized by several different factors, e.g., different objects that need to be collected. Thus, in chapter 3, we decompose the reward function into various simple factors and learn a separate value function for each component. Since each individual value function typically only depends on a subset of all features, they can be approximated more easily by a low-dimensional representation. Consequently, we find that the aggregated solution scales more efficiently than standard deep RL algorithms with respect to problem complexity.

Next, in chapter 4, we consider the noisy reward problem, i.e., when the reward the agent receives in some/all states is stochastic. To improve the stability of learning a long term value function in the noisy reward regime, we propose to train the value function using a learned estimate of the immediate reward in place of the traditional sampled reward. We demonstrate that this simple decomposition drastically improves sample efficiency under stochastic rewards in both the tabular and deep RL settings.

We then generalize reward estimation to value functions, in chapter 5, by using short-term values as a training signal for longer-term values. Specifically, our method decomposes a value function into a series of components based on the discount factor. The separation of value functions provides enhanced interpretability by querying the value function at each associated discount factor. It also improves scalability to longer-term value functions by modularly increasing the effective horizon once enough data is available. Finally, since each value function can be trained with its own respective hyperparameters (e.g, the learning rate), the result is typically a more sample efficient system.

In chapter 6, we unify the methods from chapters 3 and 5, under one general framework. Specifically, we prove under which conditions value based decompositions are equivalent to baseline approaches. Crucially, we highlight that one of the possible ways to break the equivalence is to use a different learning rate for each of the decomposed value functions.

To this end, in chapter 7, we propose a novel adaptive optimizer for deep RL that tunes the learning rate, for each learnable parameter throughout training. In certain settings, it can be interpreted as using a per state learning rate based on a partial model of the transition dynamics. Our theoretical findings demonstrate that including this adaptive tuning is comparable to normal TD learning if the optimal learning rate is found for both methods via a hyperparameter search. Further, in deep RL experiments, our findings suggest that TDprop is competitive with state of the art adaptive optimizers.

Chapter 2

Technical Background

Reinforcement learning can be analyzed through the use of Markov decision processes (MDP) (Bellman, 1957). An MDP is defined as a 5-tuple $(\mathcal{S}, \mathcal{A}, P, r, \gamma)$, with state space $\mathcal{S}$, action space $\mathcal{A}$, transition probabilities $P : \mathcal{S} \times \mathcal{A} \to dist(\mathcal{S})$ mapping state-action pairs to distributions over next states, reward function $r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$, and discount factor $\gamma \in [0, 1)$.

In this thesis, we make the simplifying assumption that $\mathcal{S}$ and $\mathcal{A}$ are finite. At every time-step t, an agent is in a state s_t, takes an action a_t, receives a reward $r(s_t, a_t)$, and transitions to the next state in the system $s_{t+1} \sim P(\cdot \mid s_t, a_t)$.

The goal of the agent is to learn a policy $\pi : \mathcal{S} \to dist(\mathcal{A})$ that maximizes the expected sum of discounted rewards from any given state. The main assumption of an MDP is the Markov assumption, i.e., that given a sequence of states and actions $(s_t, a_t, s_{t+1}, a_{t+1}, ..., s_{t+n}, a_{t+n})$, the probability of transitioning to s_{t+n+1} is equal to $p(s_{t+n+1}|s_t, a_t)$ and is not conditioned on any previous states or actions.

2.1 Policy Evaluation

The following section is concerned with determining the value of a particular policy. Specifically, we can define the value of a policy $v^\pi : \mathcal{S} \to \mathbb{R}$ as the expected discounted sum of future rewards from a particular state:

$$v^\pi(s) = \mathbb{E}^\pi \left[\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t) \middle| s_0 = s \right]. \tag{2.1}$$

Similarly, we can define the action-value function, i.e., q-function, $q^\pi : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ as the discounted sum of future rewards from a particular state given an action:

$$q^\pi(s,a) = \mathbb{E}^\pi \left[\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t) \middle| s_0 = s, a_0 = a \right]. \tag{2.2}$$

Thus, for any policy π we have the following connection between v^π and q^π :

$$v^\pi(s) = \sum_a \pi(s,a) q^\pi(s,a). \tag{2.3}$$

To solve for v^π or q^π, we can use the *expected Bellman equations* (Bellman, 1957) to make use of the recursive nature of the value functions:

$$v^\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \left(r(s,a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s,a) v^\pi(s') \right), \tag{2.4}$$

similarly for q^π we have:

$$\begin{aligned} q^\pi(s,a) &= r(s,a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s,a) v^\pi(s') \\ &= r(s,a) + \gamma \sum_{s' \in \mathcal{S}} \sum_{a' \in \mathcal{A}} \pi(a'|s') p(s'|s,a) q^\pi(s',a'), \end{aligned} \tag{2.5}$$

which can both be solved directly using matrix-inversion. To see how, we note that we can rewrite the Bellman equations as a system of linear equations in matrix form:

$$v^\pi = r^\pi + \gamma P^\pi v^\pi, \tag{2.6}$$

where $r^\pi \in \mathbb{R}^{|\mathcal{S}|}$ with $r^\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) r(s,a)$ being the expected reward vector under π, $P^\pi \in \mathbb{R}^{|\mathcal{S}| \times |\mathcal{S}|}$ with $P^\pi(s,s') = \sum_{s' \in \mathcal{S}} \sum_{a \in \mathcal{A}} \pi(a|s) p(s'|s,a)$ being the transition probability matrix induced by the policy, and $v^\pi \in \mathbb{R}^{|\mathcal{S}|}$ being the value function vector. Alternatively, by rearranging terms, we get:

$$v^\pi = (I - \gamma P^\pi)^{-1} r^\pi, \tag{2.7}$$

where I is the identity matrix. Analogously when solving for q^π we have:

$$\begin{aligned} q^\pi &= r + \gamma P'^\pi q^\pi \\ &= (I - \gamma P'^\pi)^{-1} r, \end{aligned} \tag{2.8}$$

where $q^\pi \in \mathbb{R}^{|\mathcal{S}||\mathcal{A}|}$ is the action value function in vector form, $r \in \mathbb{R}^{|\mathcal{S}||\mathcal{A}|}$ is the expected reward vector for each state and action, and $P'^\pi \in \mathbb{R}^{(|\mathcal{S}||\mathcal{A}|)\times(|\mathcal{S}||\mathcal{A}|)}$ with $P'^\pi(s,a,s',a') = \sum_{s'\in\mathcal{S}}\sum_{a'\in\mathcal{A}} p(s'|s,a)\pi(a'|s')$ being the probability over next state and action values under the current policy. We now focus solely on estimating v^π, since the results immediately extend to q^π by replacing the stochastic matrix P^π by P'^π. For convenience, we drop the superscript π and assume that $P = P^\pi$ and $r = r^\pi$ unless otherwise stated.

The solution, for v, is well defined if $(I - \gamma P)^{-1}$ is invertible. This can be shown using corollary $C.4$ from Puterman (1994) that states that if the spectral radius, $\rho(A) < 1$ then $(I - A)^{-1}$ exists. Thus, we need to show that $\rho(\gamma P) < 1$. We follow a similar analysis to section 2.1, from Bacon (2018).

To show that $\rho(\gamma P) < 1$, we first need to arm ourselves with the definition of the spectral radius.

Definition 2.1.1. *(spectral radius) The spectral radius of a matrix $A \in \mathbb{C}^{n\times n}$ is defined as:*

$$\rho(A) = \max\left\{\left|\lambda_1^A\right|, \left|\lambda_2^A\right|, ..., \left|\lambda_m^A\right|\right\}, \tag{2.9}$$

where λ^A are the eigenvalues of the matrix A. Equivalently, using Gelfand's formula (Gelfand, 1941), using any matrix norm $\|\cdot\|$:

$$\rho(A) = \lim_{k\to\infty} \left\|A^k\right\|^{\frac{1}{k}}. \tag{2.10}$$

Throughout this thesis, when referencing matrix norms, we will be referring to the infinity norm, which we now define.

Definition 2.1.2. *(infinity norm) The infinity norm for matrices is defined as:*

$$\|A\| := \max_{x\in\mathbb{R}^d, x\neq 0}\left(\frac{\|Ax\|}{\|x\|}\right) = \max_i \sum_j |A_{i,j}|, \tag{2.11}$$

which is induced by the vector norm $\|x\| := \max_i |x_i|$.

The use of the infinity norm is due to its ease of computation, it is simply the maximum absolute row sum of a matrix. Using the infinity norm and plugging γP into Gelfand's formula gives us:

$$\rho(\gamma P) = \lim_{k\to\infty} \left\|(\gamma P)^k\right\|^{\frac{1}{k}} = \gamma, \tag{2.12}$$

where we used the fact that $\|P\| = 1$, since the sum of every row of a stochastic matrix is 1, and the fact that the product of two stochastic matrices is a stochastic matrix. Therefore, by corollary $C.4$ from Puterman (1994) we have that $\rho(\gamma P) < 1$ and $(I - \gamma P)^{-1}$ exists. Before

we continue, we note a convenient property of the spectral radius, which will help us in later proofs.

Lemma 2.1.1. *(exercise 1.3.2 of Varga (1962)) For a matrix $A \in \mathbb{C}^{n\times n}$,*

$$\rho(A) \leq \|A\| . \tag{2.13}$$

Thus, whenever confronted with the problem of determining whether the spectral radius of some matrix is less than some constant, we can simply evaluate the infinity norm.

Instead of solving for the value function in closed form, which can be expensive due to the matrix inversion, we can solve for the value function iteratively using dynamic programming (Bellman, 1957) and repeatedly applying the *expected Bellman equation*:

$$v_{t+1} = r + \gamma P v_t, \tag{2.14}$$

where t is the number of updates that have been performed. We define the solution to this system as v^π, where for v^π we have $(I - \gamma P)v^\pi - r = 0$. There are several ways to determine whether this iterative scheme converges to v^π. It can be shown that equation (2.14) defines a *contraction mapping* and that under *Banach's fixed point theorem* (Banach, 1922), repeatedly applying the contraction mapping results in convergence to the solution, i.e., the fixed point. We opt for a different approach based off the theory of convergent matrices (Varga, 1962). We begin by defining the error vector as $e_t = v_t - v^\pi$ and deriving the following recursion:

$$\begin{aligned} e_t &= v_t - v^\pi \\ &= r + \gamma P v_{t-1} - v^\pi + \underbrace{((I - \gamma P)v^\pi - r)}_{=0} \\ &= \gamma P(v_{t-1} - v^\pi) = \gamma P(e_{t-1}) = (\gamma P)^t e_0. \end{aligned} \tag{2.15}$$

Thus, the error at time-step t depends on the initial error at time-step 0 and the matrix γP. Specifically, to have asymptotic convergence to v^π we require that e_t goes to 0 as $t \to \infty$, which we now define formally.

Definition 2.1.3. *(convergent matrix: definition 1.9 (Varga, 1962)) Let $A \in \mathbb{C}^{n\times n}$. Then A is convergent (to zero) if the sequence of matrices $A, A^2, A^3, ...$ converges to the null matrix 0, and is divergent otherwise.*

Conveniently, we have the following theorem that gives us the necessary and sufficient conditions for a matrix to be convergent.

Theorem 2.1.1. *(convergent matrix requirement: theorem 1.10 (Varga, 1962)) If $A \in \mathbb{C}^{n\times n}$, then A is convergent if and only if $\rho(A) < 1$.*

Thus, to show that the iterative scheme in equation (2.14) converges asymptotically we need that $\rho(\gamma P) < 1$. Conveniently, we already showed that by using Gelfand's formula we have that $\rho(\gamma P) = \gamma < 1$.

The spectral radius plays an important role in not only determining convergence but also in determining the convergence speed. We now define the *global asymptotic average rate of convergence* and show its connection to the spectral radius.

Definition 2.1.4. *(global asymptotic average rate of convergence (Puterman, 1994)) Given the recursion of error vectors e_t, the global asymptotic average convergence rate is defined as:*

$$\lim_{t\to\infty} \max_{e_0\neq 0} \left(\frac{\|e_t\|}{\|e_0\|}\right)^{\frac{1}{t}}. \tag{2.16}$$

The interpretation of the global asymptotic average rate of convergence is that it measures, in the worst case, how fast the error reduces relative to the initial error. The closer the rate is to 1 the slower the convergence (Puterman, 1994). Moreover, the connection to the spectral radius can be seen with the following derivation from Schoknecht and Merke (2003), using $e_t = (\gamma P)^t e_0$ from equation (2.15) we have:

$$\begin{aligned}
\lim_{t\to\infty} \max_{e_0\neq 0} \left(\frac{\|e_t\|}{\|e_0\|}\right)^{\frac{1}{t}} &= \lim_{t\to\infty} \max_{e_0\neq 0} \left(\frac{\|(\gamma P)^t e_0\|}{\|e_0\|}\right)^{\frac{1}{t}} && \text{(equation (2.15))} \\
&= \lim_{t\to\infty} \left(\left\|(\gamma P)^t\right\|\right)^{\frac{1}{t}} && \text{(definition of induced matrix norm)} \\
&= \rho(\gamma P). && \text{(Gelfand's formula)} \qquad (2.17)
\end{aligned}$$

Since $\rho(\gamma P) = \gamma$, the convergence rate, in this case, is determined by the discount factor. Values for γ close to one will converge slower than values close to zero. A large value for γ implies that the value function incorporates very distant rewards and thus would need more iterations for the information to propagate. We focus on the discount factor, and propose methods to learn value functions efficiently for large γ in chapter 5.

2.2 Multi-Step Methods

Multi-step methods will be used frequently throughout this thesis for their potential gains in sample efficiency (Sutton and Barto, 2018). We begin by rewriting the expected Bellman

equation in operator form:

$$\mathcal{T}_1 v = r + \gamma P v, \tag{2.18}$$

where we use operator notion $\mathcal{T} : \mathbb{R}^{|S|} \to \mathbb{R}^{|S|}$. We recall that at the solution, i.e., the fixed point, $\mathcal{T}_1 v^\pi = r + \gamma P v^\pi = v^\pi$, plugging this in to the right hand side of the one-step Bellman operator gives us multi-step methods:

$$\begin{aligned} \mathcal{T}_2 v &= r + \gamma P \mathcal{T}_1 v = r + \gamma P r + \gamma^2 P^2 v \\ \mathcal{T}_n v &= \sum_{k=0}^{n-1} (\gamma^k P^k) r + \gamma^n P^n v. \end{aligned} \tag{2.19}$$

Analogous to the one-step case, we have that at the fixed point $\mathcal{T}_n v^\pi = \sum_{k=0}^{n-1} (\gamma^k P^k) r + \gamma^n P^n v^\pi = v^\pi$. Thus, after applying the n-step operator t times we have the following recursive formulation for the error vector $e_t = v_t - v^\pi$:

$$\begin{aligned} e_t &= v_t - v^\pi \\ &= \sum_{k=0}^{n-1} (\gamma^k P^k) r + \gamma^n P^n v_{t-1} - v^\pi + \underbrace{\left((I - \gamma^n P^n) v^\pi - \sum_{k=0}^{n-1} (\gamma^n P^n) r \right)}_{=0} \\ &= \gamma^n P^n (v_{t-1} - v^\pi) = \gamma^n P^n (e_{t-1}) = \left(\gamma^k P^k\right)^t e_0. \end{aligned} \tag{2.20}$$

Therefore, $\rho(\gamma^n P^n) = \gamma^n < 1$, and the iterative system is convergent by theorem 2.1.1.

Another multi-step method is to use a geometric (λ) weighting over n-step operators (Watkins, 1989):

$$\mathcal{T}_\lambda v = (1 - \lambda) \sum_{n=1}^{\infty} \lambda^n \mathcal{T}_n v, \tag{2.21}$$

or equivalently written as,

$$\mathcal{T}_\lambda v = v + (I - \gamma \lambda P)^{-1} (r + \gamma P v - v), \tag{2.22}$$

where the full proof of the equivalency can be seen in proposition 2.8 of Bacon (2018). Similar to before, we have that at the solution $\mathcal{T}_\lambda v^\pi = v^\pi + (I - \gamma \lambda P)^{-1} (r + \gamma P v^\pi - v^\pi) = v^\pi$ and

the following recursion for the error vector:

$$\begin{aligned}
e_t &= v_t - v^\pi \\
&= v_{t-1} + (I - \gamma\lambda P)^{-1}(r + \gamma P v_{t-1} - v_{t-1}) - v^\pi - \underbrace{(I - \gamma\lambda P)^{-1}(r + \gamma P v^\pi - v^\pi)}_{=0} \\
&= (I - \gamma\lambda P)^{-1}(\gamma P - \gamma\lambda P)(v_{t-1} - v^\pi) \\
&= (I - \gamma\lambda P)^{-1}(\gamma P - \gamma\lambda P)e_{t-1} = \left((I - \gamma\lambda P)^{-1}(\gamma P - \gamma\lambda P)\right)^t e_0
\end{aligned} \tag{2.23}$$

Moreover, for $0 \leq \gamma < 1$ and $0 \leq \lambda \leq 1$ we have the following known result (Bertsekas and Tsitsiklis, 1995; Bacon, 2018):

$$\begin{aligned}
\left\|(I - \gamma\lambda P)^{-1}(\gamma P - \gamma\lambda P)\right\| &\leq \left\|(I - \gamma\lambda P)^{-1}\right\| \|\gamma P - \gamma\lambda P\| && \text{(submultiplicativity)} \\
&= (\gamma - \gamma\lambda)\left\|\sum_{n=0}^{\infty}(\gamma\lambda P)^n\right\| && \text{(Neumann series expansion)} \\
&\leq (\gamma - \gamma\lambda)\sum_{n=0}^{\infty}(\gamma\lambda)^n \|P^n\| && \text{(triangle inequality)} \\
&= \frac{\gamma - \gamma\lambda}{1 - \gamma\lambda} \leq \gamma,
\end{aligned} \tag{2.24}$$

which implies, from lemma 2.1.1, that $\rho\left((I - \gamma\lambda P)^{-1}(\gamma P - \gamma\lambda P)\right) < 1$ and the iterative scheme is convergent.

Interestingly, when compared to the one-step operator, the n-step operator has a faster convergence rate since $\gamma^n \leq \gamma$. Similar to the n-step case, the convergence rate decreases with increasing λ. However, increasing n requires more computation, since the Bellman equation needs to be unrolled n times. Similarly, with λ-returns either an infinite sum needs to be computed using the Neumann series expansion (Puterman, 1994), or a matrix inverse needs to be performed. Most importantly, however, the use of multi-step methods for our purposes is for the reinforcement learning setting and not for directly solving MDPs. As we will see, once noise is introduced in the reinforcement learning setting, the choice of n or λ plays a crucial role in the overall sample efficiency of the system.

2.3 Control

In this section, we present two classical methods for learning an optimal policy in an MDP: *value iteration* and *policy iteration*. As we will see, both methods rely on estimating a value function. While the focus of this thesis will be to learn value functions as efficiently as

possible, it will also be worth understanding how value functions are used to determine the optimal policy.

To determine the value of the optimal policy, v^*, we instead use the *Bellman-optimality operator*:

$$\mathcal{T}^* v(s) := \max_{a \in \mathcal{A}} \left(r(s,a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s,a) v(s') \right), \tag{2.25}$$

where we define $\mathcal{T}^* v(s)$ over each state individually and we use the notation $\mathcal{T}^* v$ to refer to applying the operator to every state.

In the value iteration algorithm, we repeatedly apply the Bellman optimality operator over all states, until the change in value for all states is less than some predefined threshold. The full details of value iteration can be seen in algorithm 1.

Algorithm 1 *Value Iteration*

Require: ϵ: Stopping Threshold
  $v_0(s) \leftarrow 0 \quad \forall s$
  $t \leftarrow 0$
  while $\max |v_t - v_{t-1}| > \epsilon$ or $t = 0$ **do**
    for $s \in S$ **do**
      $v_{t+1}(s) \leftarrow \max_{a \in \mathcal{A}} \left(r(s,a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s,a) v_t(s') \right)$
    end for
    $t \leftarrow t + 1$
  end while

Moreover, when the algorithm terminates we have the corresponding deterministic policy:

$$\pi(s) = \arg\max_{a \in \mathcal{A}} \left(r(s,a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s,a) v^*(s') \right), \tag{2.26}$$

which satisfies the Bellman optimality equations and is optimal (Puterman, 1994).

Since the above operator is non-linear (due to the max operation), to show convergence we cannot rely on the theory of convergent matrices but instead use Banach's fixed point theorem (Banach, 1922) for contraction mappings over Banach spaces. For our purposes, it is sufficient to consider the space over value functions, since $\mathbb{R}^n$ is a Banach space (Puterman, 1994).

Definition 2.3.1. *(contraction mapping) An operator $\mathcal{T} : \mathbb{R}^n \to \mathbb{R}^n$ is a contraction mapping in $\|\cdot\|$ if for all $u, v \in \mathbb{R}^n$ there exists a γ, $0 \leq \gamma < 1$ such that:*

$$\|\mathcal{T} v - \mathcal{T} u\| \leq \gamma \|v - u\|. \tag{2.27}$$

With the definition of contraction mappings we can now present Banach's fixed point theorem, which gives us the theoretical tools to determine that iteratively applying a contraction mapping is convergent.

Theorem 2.3.1. *(Banach's fixed point theorem (Banach, 1922)) Suppose $\mathcal{T} : \mathbb{R}^n \to \mathbb{R}^n$ is a contraction mapping. Then:*

1. *There exists a unique $v^* \in U$ such that $\mathcal{T}v^* = v^*$.*
2. *For arbitrary $v_0 \in U$, the sequence defined by:*

$$v_{t+1} = \mathcal{T}v_t = \mathcal{T}^{t+1}v_0 \tag{2.28}$$

converges to v^*.

The proof is provided in theorem 6.2.3 of Puterman (1994).

We now prove that $\mathcal{T}^*$ is a contraction mapping. For any two value functions v and v' and using the infinity norm we have:

$$\begin{aligned}
&\|\mathcal{T}^*v - \mathcal{T}^*v'\| = \max_s |\mathcal{T}^*v(s) - \mathcal{T}^*v'(s)| \\
&= \max_s \left| \max_a \left(r(s,a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s,a)v(s') \right) - \max_a \left(r(s,a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s,a)v'(s') \right) \right| \\
&\leq \max_s \max_a \left| \gamma \sum_{s' \in \mathcal{S}} p(s'|s,a)v(s') - \gamma \sum_{s' \in \mathcal{S}} p(s'|s,a)v'(s') \right| \\
&= \max_s \max_a \gamma \sum_{s' \in \mathcal{S}} p(s'|s,a)|v(s') - v'(s')| \\
&\leq \max_s \gamma |v(s) - v'(s)| \\
&= \gamma \|v(s) - v'(s)\| .
\end{aligned} \tag{2.29}$$

Thus, by definition 2.3.1, $\mathcal{T}^*$ is a contraction mapping and theorem 2.3.1 applies.

In policy iteration (Howard, 1960), instead of only performing one iteration of policy evaluation, at every step of the algorithm we find the exact value of the current policy. Specifically, the policy iteration algorithm determines the next policy π_{t+1} by first fully learning the value for the current policy v^π (either using the closed form or iterative solutions) and then applying the following operation:

$$\pi_{t+1}(s) = \arg\max_a \left(r(s,a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s,a)v^{\pi_t}(s') \right). \tag{2.30}$$

The full details of policy iteration can be seen in algorithm 2.

Algorithm 2 *Policy Iteration*

$\pi_0(s) \leftarrow 0 \quad \forall s$
$t \leftarrow 0$
while $\pi_t(s) \neq \pi_{t-1}(s) \quad \forall s$ or $t = 0$ **do**
  $v^{\pi_t} \leftarrow (I - \gamma P^{\pi_t})^{-1} r^{\pi_t}$
  for $s \in \mathcal{S}$ **do**
    $\pi_{t+1}(s) \leftarrow \arg\max_a \left(r(s,a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s,a) v^{\pi_t}(s')\right)$
  end for
  $t \leftarrow t + 1$
end while

Convergence can be shown through the *policy improvement theorem* (Sutton and Barto, 2018), which relies on the fact that at each iteration the new policy is an improvement over the previous policy, with equality at the optimal policy. Now that we have presented methods for learning optimal value functions and policies, we move on to the RL setting where the agent does not have access to the underlying reward function and transition dynamics.

2.4 Parameterization of the Value Function and Policy

We recall that in RL, instead of assuming that the reward function and transition dynamics are known to the agent beforehand, the agent is tasked with finding the optimal policy through interacting with an environment. A major choice that concerns an RL practitioner is the type of parameterization that is used for the estimate of the value function (and/or policy). The correct choice depends on the complexity of the problem at hand, as well as the domain knowledge that is available beforehand.

2.4.1 Linear Parameterization

The linear representation of the value function, often called *linear function approximation* (Sutton, 1996), takes the following form:

$$\hat{v}(s;\theta) := \langle \theta, x(s) \rangle$$

where $< \cdot, \cdot >$ denotes the inner product, $\theta \in \mathbb{R}^d$ are the trainable parameters and $x : \mathcal{S} \to \mathbb{R}^d$ is a feature mapping that extracts the feature vector from a particular state. We say that x is a *tabular representation* if it maps directly to the one-hot vector representing the state. In this case, each element i of the parameter vector θ is the value of state i. In most high

dimensional problems, however, a tabular representation is severely limiting as it lacks the ability to generalize knowledge to similar states. Moreover, assuming a lack of domain knowledge to hand craft a strong feature map, linear function approximation can also be a fairly restrictive model choice as the solution is limited to the representable space of the function approximator (Sutton and Barto, 2018).

2.4.2 Deep Representations

Using a deep neural network as a function approximator to effectively learn the feature mapping from data has been shown to be a powerful tool for RL (Mnih et al., 2015). In this section, we explore the two types of neural networks that will be used throughout this thesis; *feed forward neural networks* and *convolutional neural networks*.

Feed Forward Neural Networks: Feed forward neural networks (Rosenblatt, 1958), often referred to as multi-layer perceptrons (MLPs), learn weights for several layers of neurons that are fully connected via non-linear activation functions. An MLP has multiple (k) layers, where each layer j has m_j units. In order to compute the output of layer j, given the input h_j, we simply iterate through the layers of our MLP sequentially by computing:

$$h_{j+1} = f_j(W_j^\top h_j + b_j), \tag{2.31}$$

where $W_j \in \mathbb{R}^{m_j \times m_{j+1}}$ is the weight matrix for layer j and $f_j : \mathbb{R}^{m_{j+1}} \to \mathbb{R}^{m_{j+1}}$ is the activation function for that layer. The last layer produces the prediction $\hat{y}_i = h_k = f_{k-1}(W_{k-1}^\top h_{k-1} + b_{k-1})$. Typically, the first layer of the network is referred to as the *input layer* and does not contain any weights, i.e., $h_0 = x$. The final layer of the network is termed the *output layer*. All other intermediary layers are referred to as *hidden layers*.

Convolutional Neural Networks (CNNs) : CNNs (LeCun et al., 1989) are inspired by the neurological experiments conducted by Hubel and Wiesel (1968), where cells were discovered to be sensitive to spatial sub-regions of the visual input. CNNs learn spatially connected features by decomposing the traditional hidden layer of MLPs into a set of k, two dimensional filters that each get convolved with the input:

$$h_{j+1}^i = f_j(w_j^i * h_j + b_j), \tag{2.32}$$

where w_j^i is the ith filter of the jth layer, h_{j+1}^i the resulting feature map of the ith filter, and $*$ denotes the convolution operator.

The convolution operator is defined as follows:

$$O_{m,n} = W * H = \sum_{u=0}^{k} \sum_{v=0}^{j} W_{u,v} H_{m-u,n-v}, \tag{2.33}$$

where $W \in \mathbb{R}^{k \times j}$ is a single filter, $H \in \mathbb{R}^{m \times n}$ is the input, and $O \in \mathbb{R}^{m \times n}$ is the output of the convolution.

CNNs are typically used on tasks where spatial information at the input level is important, e.g., in image based tasks. As before with MLPs, convolutional layers can be stacked one on top of the other and connected via non-linear activation functions. While many activation functions exist in the deep learning literature, the *rectified linear* and *softmax* activations are the two activations needed for the work that follows.

Rectified-linear (ReLU) activation: The ReLU activation function (Nair and Hinton, 2010) is a popular choice for intermediary layers of the network. It is very similar to a simple linear activation $f(x) = x$, but forces the output to be positive:

$$f(x) = \max(x, 0). \tag{2.34}$$

Softmax activation: The softmax activation is generally used as the activation function for the final layer of a neural network when a probabilistic mapping that sums to one is desired, e.g., the output of the policy network in policy gradient methods (Mnih et al., 2016). The softmax function is defined as:

$$f(x)_i = \frac{e^{x_i}}{\sum_j e^{x_j}}, \tag{2.35}$$

where x_i and $f(x)_i$ is the ith value of the input x and output $f(x)$ respectively.

2.5 Stochastic Policy Evaluation

Once the parameterization has been chosen, the problem of learning the parameters can be addressed. In particular, this section examines how an RL agent can learn the value of a particular policy in the RL setting. We start by defining the full discounted return starting at time-step t, i.e., the *Monte Carlo* return, as:

$$G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k}, \tag{2.36}$$

where $r_{t+k} \sim r(s_{t+k}, a_{t+k})$ is the sampled reward at time-step t. Recall, that the value of a policy for a particular state is defined as the expected discounted sum of rewards from that state, i.e., the expectation of the Monte Carlo return. At first glance, it may seem that the infinite sum in equation (2.36) cannot be fully evaluated, since we would need to sample infinitely many times from the environment. However, we will be evaluating RL agents in the episodic setting. In such settings, we assume that the agent enters a terminal state in a finite number of steps and the episode is restarted thereafter.

One popular method for theoretically handling terminal states is to assume that they are *absorbing* states, in which they permanently *self-loop* and receive a reward of zero in perpetuity (Puterman, 1994). Conveniently, the expected return is maintained for every state. However, as we mentioned, the agent does not remain in these absorbing states but rather is transported back to one of the potential start states. From a theoretical perspective this is problematic, as we will see, the convergence proofs that we use are based off of the existence of a unique limiting stationary distribution, $\mu^\top = \mu^\top P$ (that satisfies $\lim_{N\to\infty} \sum_{t=0}^{N-1} P^t = I\mu^\top$ (Cinlar, 2013)), induced by the policy and the underlying MDP. Moreover, in MDPs with absorbing states, the stationary probability of a being in a particular state s_i is $\mu_i = 0$ for non absorbing states, which does not accurately represent the distribution of samples the agent is experiencing.

Thankfully, there is an alternative interpretation based off of per-transition discounting that fully preserves the value expectations as well as unifying the non-episodic (continuing) and episodic scenarios. Specifically, in White (2017), transitions back to start states are added and a discount factor of zero is used for those transitions. In this way, the agent transitions back to a start state, the value function is preserved for every state, the contraction properties previously presented with a constant discount factor in section 2.1 are also preserved, and most importantly, the stationary distribution is appropriately defined. We do not make a distinction in our notation for the use of transition based discounting, since it has no impact on any of the experimental or theoretical details.

Thus, an estimate $\hat{v} = (\cdot; \theta)$, parameterized by $\theta = (\theta_1, \theta_2, ..., \theta_d)$, can be trained by iteratively sampling the Monte Carlo return and regressing towards the sampled target via *stochastic approximation* (Robbins and Monro, 1951; Benveniste et al., 2012). Specifically, in stochastic approximation, updates can be performed via the following:

$$\theta_{t+1} = \theta_t + \alpha_{t+1} g(\theta_t), \tag{2.37}$$

where $\theta \in \mathbb{R}^d$ is the parameter vector, α is the learning rate, and $g(\theta)$ is the *update function* that defines how the parameters are updated.

When regressing towards the Monte Carlo return, the update function is defined as:

$$g_{MC}(\theta_t) := -\nabla_\theta \left((G_t - v(s_t; \theta_t))^2 \right). \tag{2.38}$$

The convergence properties are typically analyzed based off the expected update $\bar{g}$ under the stationary distribution:

$$\bar{g}_{MC}(\theta_t) := \mathbb{E}^\mu \left[-\nabla_\theta \left((G_t - v(s_t; \theta_t))^2 \right] \right). \tag{2.39}$$

We say that such an update function, one that is defined as sampling the negative gradient of a function, is an instance of *stochastic gradient descent* (SGD) (Widrow and Hoff, 1960).

The Monte Carlo return may have large variance caused by sampling from the policy, the environment dynamics, and the reward function. Fortunately, as previously alluded to, we can use Bellman equations (Bellman, 1957) and exploit the recursive nature of the value function. In practice, for a specific value of n, the agent will take n steps in the environment, receive n rewards, and replace the remainder of the return with its current estimate of the value function $\hat{v}$:

$$G_{t:t+n} := \sum_{k=0}^{n-1} \gamma^k r_{t+k} + \gamma^n \hat{v}(s_{t+n}; \theta_t), \tag{2.40}$$

In many cases, small values of n may have significantly less variance than larger values at the expense of added bias introduced by $\hat{v}$. We note, however, that small values of n may actually have more variance than larger values due to the *covariance* between the random rewards in the return (Konidaris et al., 2011). The optimal choice of n is typically problem dependent (Sutton and Barto, 2018) and varies from one MDP to another. It can also vary during training, since as $\hat{v}$ approaches the solution, its bias is reduced (Kearns and Singh, 2000). Moreover, it can also be treated as a random variable, based off of the theory of *random stopping times* (Wessels, 1977). Further, the optimal n can also be state dependent, since states typically have different amounts of variance in their respective return (Xu et al., 2018). We will examine methods for adapting multi-step methods based off of the discount factor in chapter 5.

As previously mentioned in section 2.2, we are not limited to using a single n-step return, but can also use a geometric weighting, i.e., the λ-return:

$$G_t^\lambda = \hat{v}(s_t; \theta_t) + \sum_{k=0}^{\infty} (\gamma\lambda)^k \delta_{t+k}, \tag{2.41}$$

where $\delta_t = r_t + \gamma \hat{v}(s_{t+1}; \theta_t) - \hat{v}(s_t; \theta_t)$ is the one-step TD-error at time-step t and the hyper-

parameter $\lambda \in [0, 1]$ plays the role of controlling the bias-variance trade-off in the return. Small values of λ put a significant weight on nearby rewards (TD($\lambda = 0$) corresponding with the 1-step return), whereas a large λ puts more weight on distant rewards (TD($\lambda = 1$) being the Monte Carlo return). Moreover, in practice, the return is typically truncated to n-steps:

$$G^{\lambda}_{t:t+n} = \hat{v}(s_t; \theta_t) + \sum_{k=0}^{n-1} (\gamma\lambda)^k \delta_{t+k}, \tag{2.42}$$

where $n \in [1, \infty)$ is the truncation length.

At each time-step t, the parameters of the value function estimate can be updated via TD learning via the stochastic update from equation (2.37) and the following update function:

$$g_{TD}(\theta_t) := \delta^{\lambda}_{t:t+n} \nabla_\theta \hat{v}(s_t; \theta_t), \tag{2.43}$$

where:

$$\delta^{\lambda}_{t:t+n} = G^{\lambda}_{t:t+n} - \hat{v}(s_t; \theta_t), \tag{2.44}$$

is the TD error. Unlike the MC update in equation (2.38), the TD update in equation (2.43) is not an instance of gradient descent, but instead falls into the category of semi-gradient methods (Sutton and Barto, 2018).

Nevertheless, both MC and TD can be analyzed from a stochastic approximation perspective, and convergence guarantees can be determined under certain assumptions (Tsitsiklis and van Roy, 1997). We restate theorem 1 from Tsitsiklis and van Roy (1997), which proves convergence with probability 1 for TD(λ) with linear function approximation by framing it into the general convergence proof for stochastic approximation (theorem 17 of Benveniste et al. (2012)).

Theorem 2.5.1. *(convergence of TD(λ) with linear function approximation: theorem 1 of Tsitsiklis and van Roy (1997)) For any $\lambda \in [0, 1]$, the TD(λ) algorithm with linear function approximation converges with probability one to the fixed point, $\Pi \mathcal{T}_\lambda v^\pi = v^\pi$, where $\Pi = X(X^\top D X)^{-1} X^\top D$ is the linear projection matrix, X is the matrix formed by placing the feature vector $x(s)$ in each row, and D is the diagonal matrix with the stationary probability $\mu(\cdot)$ as its entries, under the following main assumptions:*

1. *The Markov chain has a unique stationary distribution $\mu^\top = \mu^\top P$.*
2. *The feature vectors are linearly independent.*
3. *The learning rates are positive, non increasing, predetermined, as well they satisfy $\sum_{t=0}^{\infty} \alpha_t = \infty$ and $\sum_{t=0}^{\infty} \alpha_t^2 < \infty$.*

Key to the convergence proof is the assumption that updates are performed based on the induced Markov chain that has a unique stationary distribution. The existence of such a distribution is guaranteed for *ergodic* Markov chains, i.e., when the agent can go from any one state to any other state in a finite number of steps (Puterman, 1994). The assumption on the learning rates is standard in the stochastic approximation literature and ensures convergence under noisy updates (Benveniste et al., 2012). We also note that Tsitsiklis and van Roy (1997) consider the backward-view of TD(λ) using eligibility traces (Sutton, 1988) which are based on the same underlying operator as the forward-view presented here. Since eligibility traces are not used in the deep RL literature, but solely in with linear function approximation, we omit their exposition. Finally, the convergence of TD learning with non-linear function approximation is still an open problem (Chen et al., 2019), with known counter examples (Tsitsiklis and van Roy, 1997).

2.6 Stochastic Value-Based Control

In this section, we discuss methods for learning policies through the use of q-values. We recall from section 2.3, that given the optimal q-values, the optimal policy is simply to act *greedily* with respect to the q-values, i.e., $\arg\max_a q*(s,a)$. However, in the RL setting, agents do not have access to the underlying dynamics and must, through trial and error, learn which actions are good and which are bad in a given state. In other words, to determine which action is best for a given state, many potential suboptimal actions need to be tried first.

Given q-values, the behavioral policy, i.e., the policy that is sampled to take actions in the environment, is designed to ensure exploration of the state and action space. Arguably the simplest behavioural policy is an ϵ-greedy policy (Sutton and Barto, 2018); with probability ϵ, the agent takes a random action, and with probability $(1-\epsilon)$ the agent acts greedily with respect to its q-values (i.e., $\arg\max_a \hat{q}(s,a)$). In the deep RL literature, several other notable exploration techniques have been proposed, including: pseudo-counts (Bellemare et al., 2016), randomized value functions (Osband et al., 2019), and noisy networks (Fortunato et al., 2017). While all of these methods tend to improve sample efficiency, we will mainly use ϵ-greedy exploration due to its simplicity.

The first set of algorithms that we consider are *on-policy* algorithms. We say that an algorithm is on-policy if the return, and thus the target, is generated directly from the policy that is being estimated. One example of an on-policy algorithm is *SARSA* (Rummery and Niranjan, 1994). In SARSA, the agent performs TD updates based off of the actions that are sampled from the policy and taken in the environment. Specifically, SARSA(λ) uses the

following update function:

$$g_{SARSA}(\theta_t) = \left(\sum_{k=0}^{n-1}(\gamma\lambda)^k\delta_{t+k}\right)\nabla_\theta\hat{q}(s_t, a_t; \theta_t), \tag{2.45}$$

where $\delta_t = r_t + \gamma\hat{q}(s_{t+1}, a_{t+1}) - \hat{q}(s_t, a_t; \theta_t)$. Once the parameters are updated, the new policy is immediately determined based on the new q-values. Thus, SARSA performs a policy evaluation update and then implicitly performs a policy improvement step by using the new q-values for its policy.

The convergence of SARSA to the optimal policy under a tabular representation relies on the fact that the behavioural policy is greedy in the limit, i.e., that $\epsilon \to 0$, and that every state and action pair is visited infinitely many times (Singh et al., 2000). Moreover, convergence with linear function approximation can also be shown (Perkins and Precup, 2003).

Alternatively, in *expected SARSA* the sampling over next actions is removed and the expectation is used instead:

$$g_{exp}(\theta_t) = \left(\sum_{k=0}^{n-1}(\gamma\lambda)^k\delta_{t+k} + \sum_a \pi(a|s_t)\hat{q}(s_t, a) - \hat{q}(s_t, a_t)\right)\nabla_\theta\hat{q}(s_t, a_t; \theta_t), \tag{2.46}$$

The fundamental difference between on-policy and off-policy algorithms is that in off-policy learning, the behavioural policy differs from the target policy. For example, in Q-learning (Watkins, 1989), the agent follows a exploratory behavioural policy such as ϵ-greedy, however, the learning update assumes that the best possible action is taken at the next time-step. Specifically, the parameters are updated through the following update function:

$$g_Q(\theta_t) = \left(r_t + \gamma \max_{a'}\hat{q}(s_{t+1}, a') - \hat{q}(s_t, a_t; \theta_t)\right)\nabla_\theta\hat{q}(s_t, a_t; \theta_t). \tag{2.47}$$

The convergence of q-learning has been established in the tabular setting, however, with function approximation there are known counter examples (Baird, 1995). Nevertheless, off-policy algorithms can increase sample efficiency by using data generated from different policies. One use case is that of experience replay (Lin, 1992) (used in DQN (Mnih et al., 2015)), which stores previous experiences in a buffer and replays them for updates. Moreover, while we only present the one-step version, multi-step versions, including λ-weighted versions, exist in the literature (Harutyunyan et al., 2016). We also note that most of the methods that we propose are rooted in on-policy algorithms due to their stronger theoretical guarantees (Tsitsiklis and van Roy, 1997).

2.7 Policy Gradients

In this section, we present an alternative approach to learning optimal policies using *policy-gradient* methods. Instead of defining the behavioural policy as a function of q-values, policy gradient methods (Sutton et al., 2000) directly optimize for, and sample actions from, a parameterized policy $\pi(a|s;\theta)$. Specifically, policy gradient methods maximize the expected return conditioned on the start state distribution and stationary distribution induced by the current policy.

One implementation of the policy gradient theorem, is the *REINFORCE* algorithm (Williams, 1992), which uses the full Monte Carlo return as its objective, resulting in the following update function:

$$g_{RE}(\theta_t) = \nabla_\theta \log \pi(s_t, a_t; \theta_t) G_t. \tag{2.48}$$

However, using the full Monte Carlo return results in gradient estimates with high variance. To reduce the variance, we can both use multi-step returns and subtract a state dependent baseline $b(s_t)$, where typically $b(s_t)$ is the value function $\hat{v}(s_t)$:

$$g_{ac}(\theta_t) = \nabla_\theta \log \pi(s_t, a_t; \theta_t)(G^\lambda_{t:t+n} - \hat{v}(s_t)), \tag{2.49}$$

where $\hat{v}(s_t)$ can be separately learned as described in section 2.5 with its own set of parameters. Methods that learn both a value function and a parameterized policy in this way are termed *actor-critic* methods. We also note that $G^\lambda_{t:t+n} - \hat{v}(s_t)$ is simply the TD error for λ-returns. In the context of policy gradient methods we will use $A^\lambda_{t:t+n} = G^\lambda_{t:t+n} - \hat{v}(s_t)$ and refer to it as the *advantage* function. Recently, several actor-critic deep RL methods have been proposed (Mnih et al., 2016; Schulman et al., 2017) to achieve state of the art results on several Deep RL tasks.

One popular approach, called A2C, the synchronous version of A3C (Mnih et al., 2016), will be used in several instances throughout this thesis. The main idea is to use multiple parallel threads to collect samples from the current policy. Then, one synchronous update is performed to both the policy and value function based off of the collected data. The new policy is then rolled out to all threads. We will use both A2C and A3C as a baseline method in the following chapters.

Another popular actor-critic approach is *proximal policy optimization* (PPO) (Schulman et al., 2017). PPO is based off of *trust region policy optimization* (TRPO) (Schulman et al., 2015) where the policy updates are constrained to a given optimization region (a trust region). Unlike standard actor-critic algorithms, PPO collects data and computes the advantage

under the current policy, but then performs multiple parameter updates based on a clipping objective between the current and old policy, π and π_{old}:

$$g(\theta_t^\pi) = \min\left(\frac{\pi(s_t, a_t; \theta_t^\pi)}{\pi_{old}(a_t|s_t)} A_{t:t+n}^\lambda, \text{clip}\left(\frac{\pi(s_t, a_t; \theta_t^\pi)}{\pi_{old}(a_t|s_t)}, 1-\epsilon, 1+\epsilon\right) A_{t:t+n}^\lambda\right), \tag{2.50}$$

where $\epsilon < 1$ is a hyperparameter that constrains the update and $A_{t:t+n}^\lambda = G_{t:t+n}^\lambda - \hat{v}(s_t; \theta_t^v)$ is the advantage function.

2.8 Optimization

We recall that the standard stochastic approximation update takes the following form:

$$\theta_{t+1} = \theta_t + \alpha g(\theta_t), \tag{2.51}$$

where α is the global learning rate and $g(\theta_t)$ is the update function. It is also common to use gradient averaging, also known as, momentum (Polyak and Juditsky, 1992), which uses an exponential moving average of past gradients to perform updates:

$$\bar{g}_{t+1} = \beta_1 \cdot \bar{g}_t + (1 - \beta_1) \cdot g(\theta_t) \tag{2.52}$$

A detailed version of the algorithm can be seen in algorithm 3.

Algorithm 3 *SGD*

Require: α: Learning rate
Require: $\beta_1 \in [0, 1)$: Exponential decay rate
  $\bar{g}_0 \leftarrow 0$
  function UPDATE($g(\theta_t)$)
    $\bar{g}_{t+1} \leftarrow \beta_1 \cdot \bar{g}_t + (1 - \beta_1) \cdot g(\theta_t)$
    $\theta_{t+1} \leftarrow \theta_t - \alpha \cdot \bar{g}_{t+1}$
  end function

In recent years, more advanced optimization techniques have been developed to optimize deep neural networks. Specifically, most of the deep RL work that follows uses Adam (Kingma and Ba, 2014), which is an adaptive optimization technique that maintains separate learning rates for each parameter of the model. Adam, uses gradient averaging and also keeps a moving average of the squared gradient for each parameter:

$$\theta_{t+1} = \theta_t + \frac{\alpha}{\sqrt{z_{t+1}} + \epsilon} \bar{g}_{t+1} \quad \text{where} \quad z_{t+1} = (1 - \beta_2) g(\theta_t)^2 + \beta_2(z_t), \tag{2.53}$$

where z represents the adaptive learning rate per parameter, β_2 is a tracking hyperparamter, and ϵ is a damping hyperparameter to avoid divisions by 0. A detailed description can be found in algorithm 4.

Algorithm 4 *Adam*[1]

Require: α: Learning rate
Require: $\beta_1, \beta_2 \in [0, 1]$: Exponential decay rates
Require: $\epsilon \in [0, 1)$: Damping Hyperparameter
  $\bar{g}_0 \leftarrow 0$
  $z_0 \leftarrow 0$
  function UPDATE($g(\theta_t)$)
    $\bar{g}_{t+1} \leftarrow \beta_1 \cdot \bar{g}_t + (1 - \beta_1) \cdot g(\theta_t)$
    $z_{t+1} \leftarrow \beta_2 \cdot z_t + (1 - \beta_2) \cdot g(\theta_t)^2$
    $\theta_{t+1} \leftarrow \theta_t - \alpha \cdot \bar{g}_{t+1} / (\sqrt{z_{t+1}} + \epsilon)$
  end function

[1] we omit bias corrections for conciseness.

Chapter 3

Reward Decomposition

In the following chapter, we focus on solving difficult RL problems by decomposing them into a set of simpler tasks for the RL agent to solve separately. Specifically, we study tasks where the reward function can be intuitively decomposed into several components. For example, consider the task of collecting two distinct objects, A and B. In this scenario, the decomposition of the reward function is fairly straightforward; we can split the total reward r into r^A and r^B, the rewards for collecting objects A and B respectively. This simple example of a decomposable reward function can be easily extended to more complicated settings, e.g., to different sub-goals within a single task.

If such a decomposition exists, and is given to the learner in advance, then strategies for exploiting this domain knowledge can be used in order to speed up learning. For example, we explore approximating each individual value function with only a subset of the state features, i.e., the features that the respective reward function needs to accurately represent the value function. Unfortunately, as we will explore in the following sections, state space reduction is not always possible when trying to optimize for the optimal value function.

To this end, we propose regularized solutions for training and aggregating decomposed reward functions. One of our key insights is that optimizing towards a regularized objective, while suboptimal, can lead to significant gains in sample efficiency in complicated RL tasks. We first formalize and provide theoretical properties of several strategies for decomposition. Then, we use these decompositions to show empirical sample efficiency benefits on both a simple grid world domain and Ms. Pac-Man from the ALE (Bellemare et al., 2013).

3.1 General Value Functions

HRA builds upon the *horde* architecture (Sutton et al., 2011). The horde architecture consists of a large number of *demons* that learn in parallel via off-policy learning. Each demon trains

a separate *general value function* (GVF) based on its own policy and *pseudo-reward* function. A pseudo-reward can be any feature-based signal that encodes useful information.

GVFs are defined formally as value functions over rewards that are not necessarily related to the rewards of the underlying MDP. Specifically, we have that:

$$v_{gvf}(s) = \mathbb{E}^{\pi}\left[\sum_{t=0}^{\infty} \gamma^t r_t^{gvf} | s_0 = s\right]. \tag{3.1}$$

Where we note that the original value function can be also be thought of as a GVF.

3.2 Proposed Approach

Our approach is based on the decomposition of the reward function. Specifically, we propose to decompose the reward function r into $Z + 1$ components:

$$r(s, a) = \sum_{z=0}^{Z} r^z(s, a), \tag{3.2}$$

and to train a separate RL agent on each of these reward functions. Because each component z has its own reward function, it also has its own q-value, q_z parameterized by θ^z. We refer to the combined network that represents all q-functions as the *hybrid reward architecture* (HRA).

3.2.1 Aggregation

Action selection for HRA is based on the sum of the agent's q_z values, which we call q_{HRA}:

$$q_{\text{HRA}}(s, a) := \sum_{z=0}^{Z} w_z(s) q_z(s, a; \theta^z), \tag{3.3}$$

where w_z is the weight applied to q_z. For the most direct comparison to the original objective, we mainly consider setting $w_z = 1\ \forall z$, however, we will loosen this constraint in section 3.4 for more efficient solutions based on added domain knowledge. In the following sections, we analyze the advantages and disadvantages of this aggregation by considering different objectives for training each local q-function.

3.2.2 Greedy Target

One possibility is to train each q_z towards their own locally optimal value using the standard q-learning update:

$$g_q(\theta_t) = \left(r_t^z + \gamma \max_{a'} \hat{q}_z(s', a'; \theta^z) - \hat{q}_z(s, a; \theta^z)\right) \nabla_{\theta^z} \hat{q}_z(s, a; \theta_t^z). \tag{3.4}$$

Summing the locally optimal q-values will not necessarily result in the q-values of the globally optimal policy.

To see this, let us examine an object collection task where the agent starts in the middle state, s_0, with objects to be collected directly to the left and right of the agent. There are 3 actions: $a_0 = left$, $a_1 = right$, and $a_2 = stay$. Finally, once one of the objects is collected, a positive reward of either r^A or r^B is received and the episode terminates. Our locally optimal q-function for the agent responsible for collecting object A is:

$$q_A(s_0, \cdot) = (r^A, 0, \gamma r^A), \tag{3.5}$$

where q^B is symmetric to q^A, with the q-values swapped for taking the left and right actions. Thus, the q-values for the aggregated agent are:

$$q_{HRA}^{local}(s_0, \cdot) = (r^A, r^B, \gamma r^A + \gamma r^B). \tag{3.6}$$

On the other hand, the optimal q-values for the global problem are:

$$q^*(s_0, \cdot) = (r^A, r^B, \gamma \max(r^A, r^B)). \tag{3.7}$$

As we can see, the action *stay* is being overestimated by the decomposed agent (q_{HRA}^{local}). Specifically, following the greedy policy results in the agent always taking the *stay* action if $\gamma > \frac{r^A}{1+r^B}$ and $\gamma > \frac{r^B}{1+r^A}$. This undesirable behaviour is caused by the combination of q-values that were trained with different target policies. Further analysis of this type of behaviour is provided in Laroche et al. (2017), where the term *attractor* is used to describe states that systems based on aggregated q-values are attracted towards.

3.2.3 Optimal Target

Alternatively, each q_z can be trained towards the same policy using expected SARSA (van Seijen et al., 2009):

$$g_{exp}(\theta_t) = \left(r_t^z + \gamma \sum_{a' \in \mathcal{A}} \pi(a'|s_{t+1}) \hat{q}_z(s_{t+1}, a') - \hat{q}_z(s_t, a_t) \right) \nabla_{\theta^z} \hat{q}_z(s, a; \theta_t^z). \tag{3.8}$$

In fact, as long as we update each $\hat{q}_z$ towards the same target policy, π, we can show that when using a tabular representation there is a strict equivalence at every training step, between standard expected SARSA and the summed q-values (Laroche et al., 2017). It follows directly from this result that training towards the optimal action of the summed objective, $\pi(a'|s') = \max_{a'} \sum_z q_z(s', a')$, will result in learning the optimal policy of the original (non-decomposed) MDP. For the rest of the chapter, we focus on a regularized objective as we would like to show improvement over the baseline.

3.2.4 Random Target

A sub-optimal policy that each q_z can be trained towards is the random policy:

$$g_{rand}(\theta_t) = \left(r_t^z + \gamma \frac{1}{\mathcal{A}} \sum_{a' \in \mathcal{A}} \hat{q}_z(s_{t+1}, a') - \hat{q}_z(s_t, a_t) \right) \nabla_{\theta^z} \hat{q}_z(s, a; \theta_t^z). \tag{3.9}$$

The summed q-values, which we call q_{HRA}^{rand}, will represent the action-value function under the random policy. By using this target, the agent will no longer be optimizing towards the optimal policy, except in degenerate MDPs. We note, however, that the behavioural policy can still act greedily with respect to the sum of the trained q-values, providing improvement over the random policy.

The random policy target has convenient properties in terms of state space reduction. To see this, let us reconsider our object collection example from section 3.2.2. Under the random policy, the probability that the agent obtains an object at any point in time is independent of whether or not other objects are available. This property allows us to reduce the state space for each agent, by only using the agent's position in the grid and whether the corresponding object is available or not.

On the other hand, under the optimal policy, the expected time for collecting an object depends on which other objects are still available in the world. Therefore, if we apply the same state space reductions that we previously described for q_{HRA}^{rand}, each sub-task would become *partially observable*, potentially causing convergence issues due to breaking the

Markov property.

3.2.5 Extra Domain Knowledge

In its basic setting, the only domain knowledge applied to HRA is in the form of the decomposed reward function. However, one of the strengths of HRA is that it can easily exploit more domain knowledge. Domain knowledge can be exploited in one of the following ways:

1. **Removing irrelevant features:** Features that do not affect the decomposed reward, r^z, add noise to the learning process and can be removed.

2. **Identifying terminal states:** Terminal states are states from which no further reward can be received; they have by definition a value of 0. In our use case, this corresponds to states after a particular object has been collected. The agent responsible for that object will receive a reward of 0 until the episode is reset. Using this knowledge, HRA can refrain from approximating this value by the value network, such that the weights can be fully used to represent non-terminal states.

3. **Using pseudo-reward functions:** Instead of updating an HRA head using a component of the environment reward, it can be updated using a pseudo-reward. For example, instead of receiving the environmental reward for collecting a particular object, the agent can receive rewards for navigating to the corresponding grid location. The main benefit of this approach is that value functions can still be trained even after the object has been collected. In this scenario, each value function trained using pseudo-rewards is considered a *general value function* (GVF) (Sutton et al., 2011).

3.3 Related Work

The horde architecture (Sutton et al., 2011) is focused on building up general knowledge about the world, encoded via a large number of GVFs. Unlike the horde architecture, HRA focusses on training separate components of the environment-reward function in order to more efficiently learn a control policy.

Similar to HRA, *universal value function approximators* (UVFA) (Schaul et al., 2015) builds on horde as well, but extends it along a different direction. UVFA enables generalization across different tasks/goals. It does not address how to solve a single, complex task, which is the focus of HRA.

UNREAL (Jaderberg et al., 2016) also shares similarities with HRA. Specifically, in UNREAL several different auxiliary tasks are used to learn a shared representation to improve the sample efficiency in the main task. In contrast, in HRA we decompose the problem into simple components that when aggregated yield the final solution.

There have been several works that consider decomposing the reward function while still maintaining optimality of the solution (Russell and Zimdars, 2003; Sprague and Ballard, 2003). As we mentioned in section 3.2.3, to achieve a benefit over standard approaches, these methods typically need to make even stronger assumptions about the environment. For example, if each component of the reward function is fully independent of one another, then each subtask can be solved separately. Unlike these works, we consider the scenario where the target policy is potentially sub-optimal, leading us to large reductions in the state space under less strict assumptions.

Finally, HRA relates to *options* (Sutton et al., 1999; Bacon et al., 2017), and more generally *hierarchical learning* (Barto and Mahadevan, 2003; Kulkarni et al., 2016). Options are temporally-extended actions that, like HRA's heads, can be trained in parallel based on their own (intrinsic) reward functions. Once an option has been trained, a higher-level agent that uses an option sees it as an action and evaluates it using its own reward function. This can yield great speed-ups in learning, and help substantially with exploration. Unlike options, HRA does not follow a single sub-agent for an extended period of time, but rather aggregates all of the sub-agents q-values at every time-step.

3.4 Experiments

In the following section, we demonstrate how HRA can be used to improve sample efficiency in both a toy object collection task as well as Ms. Pac-Man.

3.4.1 Object Collection Task

We first consider a task where an agent has to collect objects as quickly as possible in a 10×10 grid. There are 10 possible object locations spread out across the grid. For each episode, an object is randomly placed on 5 of those 10 locations. The agent starts at a random position and can take one of the 4 cardinal actions (left, right, up, down). It receives a reward of $+1$ if an object is collected and 0 otherwise. An episode ends after all 5 objects have been collected or after 300 steps, whichever comes first. The code for this experiment is available at github.com/Maluuba/hra.

We use DQN (Mnih et al., 2015) as our baseline, and compare it to HRA using the same

network. The network consists of a binary input layer of size 110, encoding the agent's position and whether there is an object on each possible location. This is followed by a fully connected hidden layer of size 250. For HRA, this layer is connected to 10 heads consisting of 4 nodes each, corresponding to the action-value under each individual reward function. For DQN, we use the same structure, but compute the sum over all 10 heads resulting in an output size of 4 (corresponding to the number of actions). The architectures can be seen in figure 3.1.

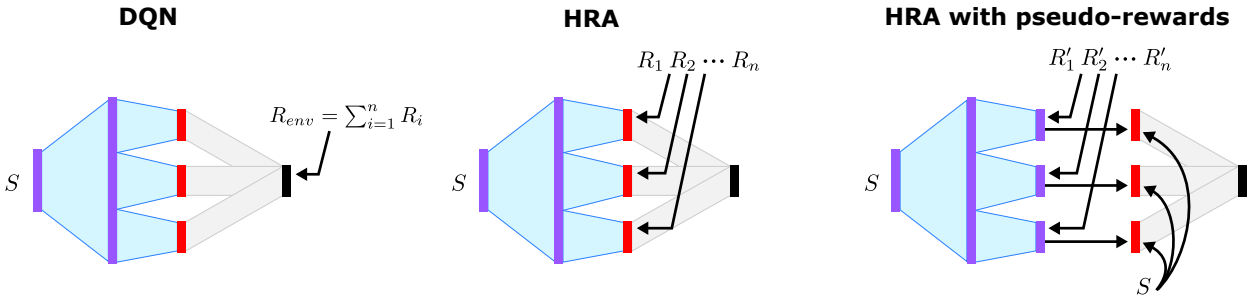

Figure 3.1: The different network architectures used in the object collection task. Left: DQN with several output heads used for DQN and DQN+1. Center: HRA architecture used for HRA and HRA+1. Right: HRA architecture with pseudo-rewards used for HRA+2 and HRA+3.

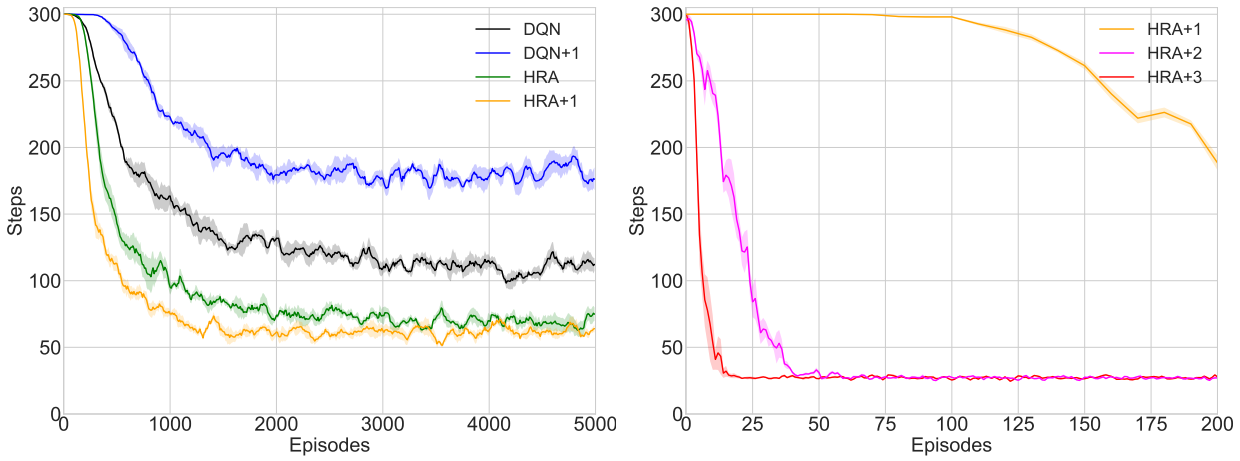

Figure 3.2: Results on the object collection domain averaged over 5 seeds. Shaded regions denote 95% confidence intervals. The agents have to collect 5 randomly placed objects. We plot the number of environment steps that each method needs to collect all the objects (y-axis) relative the number of training episodes (x-axis). We note that DQN, DQN+1, HRA, and HRA+1 are Deep RL methods, whereas HRA+2 and HRA+3 use a tabular representation that was enabled by the decomposition.

Besides for the full network, we test using the following three levels of domain knowledge, as outlined in section 3.2.5: 1) removing the irrelevant features for each head by providing only the position of the agent and the corresponding object feature; 2) the knowledge from 1 and additionally identifying terminal states; 3) the knowledge from 1 and 2 and using pseudo rewards for learning GVFs to go to each of the 10 locations. The head for a particular location copies the q-values of the corresponding GVF if the location currently contains an object, or outputs 0 otherwise. We refer to these as *HRA+1*, *HRA+2* and *HRA+3*, respectively. For DQN, we also tested a version that used the same state space reduction as *HRA+1*, we refer to this version as *DQN+1*.

Training samples are generated by a random policy and experiences are replayed periodically using a replay buffer (Lin, 1992) for learning updates. The training process is tracked by evaluating the greedy policy with respect to the learned value function after every episode. For HRA, we tested q^{local}_{HRA} as the training target (using equation (3.4)), as well as q^{rand}_{HRA} (using equation (3.8)). Similarly, for DQN, we used the default training target, q^*, as well as q^{rand}. We conducted a coarse grid search for the learning rate, $\alpha \in [0.0001, 0.0005, 0.001, 0.005]$, and the discount factor, $\gamma \in [0.8, 0.9, 0.95, 0.99]$, for each method separately and found that setting the learning rate to 0.001 and the discount factor to 0.99 worked best for both methods.

The results are shown in figure 3.2 using the best settings of each method. For DQN, using q^* as the training target resulted in the best performance, while for HRA, using q^{rand}_{HRA} resulted in the best performance. Overall, HRA shows a clear performance boost over DQN, even though the network is identical. Reducing the state space improves the performance of HRA (see HRA vs. HRA+1). However, using that same knowledge for DQN results in a decrease in performance, which we hypothesize is from the added partial observability that we discussed in section 3.2.3. The big boost in performance that occurs when the terminal states are identified is due to the representation reducing to an exact tabular representation. For the tabular methods, we used the same learning rate as the optimal learning rate for the deep network version.

3.4.2 Ms. Pac-Man

Next, we test our approach on the Atari 2600 game, Ms. Pac-Man, using the ALE (Bellemare et al., 2013). The main objective of the game is to collect different objects while avoiding being caught by the ghosts. The standard deep RL approach to solving Ms. Pac-Man is for the agent to learn a mapping from raw-pixels to actions. We implement one of the state of the art deep RL methods, *asynchronous advantage actor-critic* (A3C) (Mnih et al., 2016). It learns using a CNN as its representation, the policy gradient to learn the policy,

object	points
pellet	10
power pellet	50
1^{st} blue ghost	200
2^{nd} blue ghost	400
3^{rd} blue ghost	800
4^{th} blue ghost	1,600
cherry	100
strawberry	200
orange	500
pretzel	700
apple	1,000
pear	2,000
banana	5,000

Table 3.1: Point breakdown of objects in Ms. Pac-Man.

and multi-step returns with $n = 5$ to train both the policy and the value function. We refer to this algorithm as A3C(pixels). We use the same hyperparameters and network as the original paper (Mnih et al., 2016) with an additional search over the learning rate, $\alpha \in [0.0001, 0.00025, 0.0005, 0.00075, 0.001]$.

In order to decompose the reward function, we first extract the position of the different objects which are summarized in table 3.1. Moreover, the position and direction of Ms. Pac-Man are also needed to construct a Markov state for our decomposition. To get this information, we first remove the bottom and top of the 210×160 screen resulting in 160×160 pixels. From this, we extract the position of the different objects and create a separate input channel by encoding its location with an accuracy of 4 pixels. This results in 11 binary channels of size 40×40. Specifically, there is a channel for Ms. Pac-Man, each of the 4 ghosts, each of the 4 blue ghosts (these are treated as different objects), the fruits, plus one channel with all the pellets (including power pellets that turn the ghosts blue).

Using this domain knowledge we also construct a new version of A3C, A3C(channels), where we still train a CNN using the same objective but use the privileged features as input (the output of our object detector). Specifically, for A3C(channels), we combine the 4 channels of the ghosts into a single channel to allow it to generalise better across ghosts. We do the same with the 4 channels of the blue ghosts. Finally, instead of giving the history of the last 4 frames as done in Mnih et al. (2016) to create a Markov state, we give the orientation of Ms. Pac-Man as a 1-hot vector of length 4 (representing the 4 compass directions).

For HRA, we learn a GVF for reaching a particular point in the grid under the random

policy. Specifically, once a particular position is reached, the associated GVF receives a reward of $+1$. Moreover, since each q_z only needs access to state features that directly impact r^z, each sub-agent only needs access to the current location of Ms. Pac-Man and its current direction, resulting in approximately 950 states per GVF. Thus, due to the large state reduction, each q_z can be trained using a simple tabular representation. The GVFs are trained online with expected SARSA with $\alpha = 1$ and $\gamma = 0.99$.

For aggregation, we use the extracted positions of all the available collectable objects and reweight the GVFs based on their true point values from table 3.1. Moreover, if an object is not on the screen, we set its q-values to 0. Ghosts in Ms. Pac-Man do not have a predefined point value but rather cause a loss of life if touched. To handle this, we normalize the summed q-values of the collectable objects to be between $[0, 1]$, and then use a negative weight of -10 for the ghost's GVFs. For the behavioural policy, we act greedily with respect to the summed q-values with an added *diversification* head. Specifically, for the first 50 steps the diversification head adds $q_d \sim \mathcal{U}(0, 20)$ to the summed q-value to encourage diverse trajectories.

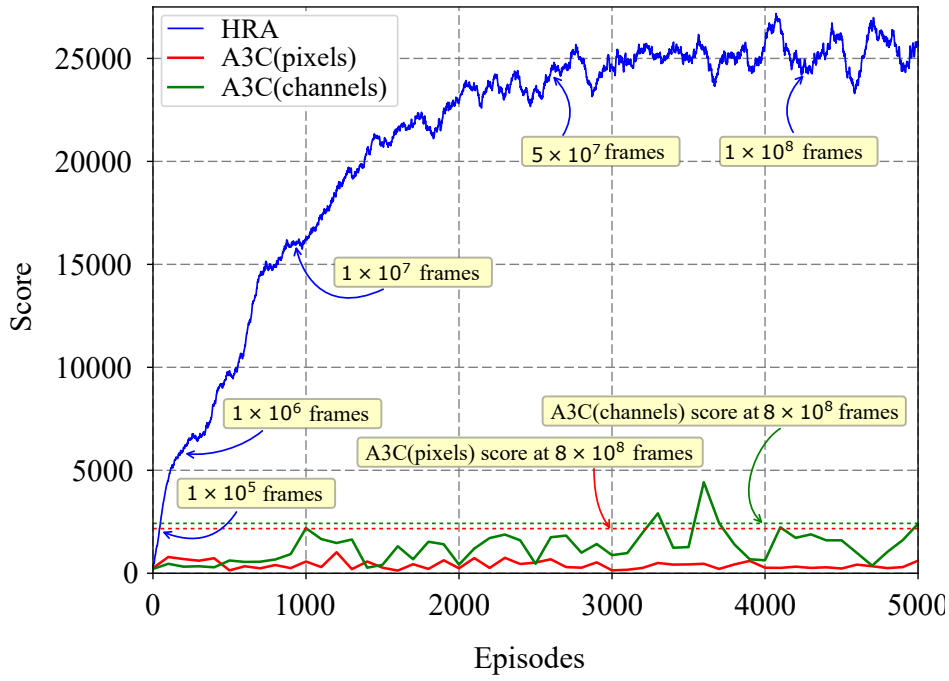

Figure 3.3: Per episode score (y-axis) plotted against number of training episodes (x-axis). Our method, HRA (in blue), outperforms both deep RL baselines. A3C(pixels) in red uses the standard ALE preprocessing. A3C(channels) uses the same privileged features (described in the text) as our approach.

Figure 3.3 shows the training curves for our method as well as the baselines. We train

both versions of A3C for 800 million frames. Because HRA learns quickly, we train it only for 5000 episodes, corresponding with about 150 million frames (note that better policies result in more frames per episode). We can see that HRA clearly outperforms both A3C(pixels) and A3C(channels). Comparing A3C(pixels) and A3C(channels) in figure 3.3 reveals a surprising result. While we use advanced preprocessing for A3C(channels) by separating the screen image into relevant object channels, this did not significantly change the final performance, although it did help it learn more quickly.

3.5 Discussion

One of the strengths of HRA is that it can exploit domain knowledge to a much greater extent than standard methods. This is clearly shown in the Ms. Pac-Man experiments, where knowledge of the reward function and the state space allowed for the decomposition of the main task into many simple (tabular) tasks. Further knowledge allowed for the use of GVFs that can be trained efficiently and aggregated using the appropriate weighting.

Another potential benefit of the weighted decomposition could be to stabilize training of neural networks. Many deep RL approaches, including A3C (Mnih et al., 2016), simply clip the reward function to be within $[-1, 1]$ to stabilize training. However, in many tasks there exist rewards of varying magnitudes which can result in the clipping heuristic simply maximizing the reward frequency (van Hasselt et al., 2016). Instead, training many deep RL agents using a clipped reward component and then applying the appropriate weighting during aggregation could result in accurately representing the value function while still maintaining stability. Similarly motivated, van Hasselt et al. (2016) propose to learn how to normalize the outputs to address varying reward magnitudes. Their approach has the benefit of potentially requiring less domain knowledge (the reward decomposition and the magnitude), however, it comes at the expense of added parameters to train.

On the other hand, the major caveats of HRA are that it requires domain knowledge and potentially reaches a sub-optimal solution. More recent approaches, such as in Schrittwieser et al. (2019), use model-based techniques for deep RL and outperformed the results presented here for Ms. Pac-Man, without using the same amount of domain knowledge or a regularized objective. In the following chapters, we explore methods that do not make use of additional domain knowledge or that restrict the solution space.

Chapter 4

Reward Estimation

The rest of this thesis explores solution methods that decompose the value estimation problem through the Bellman equation without additional domain knowledge. A crucial part of the Bellman equation is the reward component. When doing classical dynamic programming, the expected reward is assumed to be known (Bellman, 1957). However, in standard online reinforcement learning, the reward is sampled from the environment at every time-step (Sutton and Barto, 2018). This sampling procedure can result in high variance during training and slow convergence to the optimal value function. More importantly, stochastic rewards are prevalent in certain RL scenarios, especially those where the rewards are based on sensory data (Everitt et al., 2017) or are human generated (Knox and Stone, 2012).

In this chapter, we propose a simple method for updating RL algorithms to compensate for stochastic reward signals. We suggest learning an estimator for both the expected reward, and the value function, i.e., using an estimate of the reward $\hat{r}$ to update the value function $\hat{v}$ and policy π, rather than the sampled rewards. We argue that the reward function is easy to learn and a powerful variance reduction tool for training RL systems.

Crucial to the success of the approach is the fact that learning an estimate of the reward is a much easier task than learning the entire value function. We recall that in section 2.1, we discussed that the convergence rate of policy evaluation is determined by the discount factor, with smaller values of γ having a faster convergence rate. Using this logic, and the fact that reward estimation can be seen as learning a value function with $\gamma = 0$, it is not surprising that a reward estimate would indeed converge faster than a value estimate with a larger γ. We confirm the validity of the approach and show that it results in theoretical and empirical performance gains in certain tabular and function approximation settings.

4.1 Related Work

Estimating the reward function falls into the large array of work on *model-based* RL (Sutton, 1990; van Seijen and Sutton, 2013; Silver et al., 2016b; Racanière et al., 2017; Henaff et al., 2017; Feinberg et al., 2018; François-Lavet et al., 2018; Talvitie, 2018). In model-based RL, the agent learns both an estimate of the reward function and an estimate of the transition dynamics. The joint model can then be used for performing expected updates (Sutton, 1990; van Seijen and Sutton, 2013; Feinberg et al., 2018). However, in our case we do not require learning the potentially complex transition dynamics, as we explicitly aim to handle stochastic rewards. Nevertheless, reward estimation can be interpreted as a single step case of *model-based for model-free* RL, which uses a model to help the training of methods that do not usually require a model.

Other branches of RL literature also sometimes use an estimate of the reward function, such as *inverse reinforcement learning* (IRL) and *transfer learning* for RL. IRL (Ng and Russell, 2000; Abbeel and Ng, 2004) involves inferring reward functions from demonstrations given the assumption that the demonstrations were optimal under said reward function. In transfer learning for RL (Taylor and Stone, 2009), the agent is trained in one environment and then tested in another, with the underlying assumption that some components of the training task can be transferred over to the test task. In some transfer learning scenarios, the reward changes but the transition dynamics remain the same from training to testing (Laroche and Barlier, 2017; Barreto et al., 2017). In these scenarios, learning a reward function for each new task is combined with the previously learned model to generate the new value function and policy.

4.2 Proposed Approach

Under a stochastic reward $r_t \sim r(s_t, a_t)$, an additional source of variance is injected into the value function update. To reduce this variance, we introduce an estimator for the reward at a given state $\hat{r}(s_t, a_t; \theta^r_t)$. In the function approximation case, learning this reward estimator becomes a simple regression problem that can be updated via gradient descent:

$$g_{\hat{r}}(\theta^r_t) = -\nabla_{\theta^r} \left(r_t - \hat{r}(s_t, a_t; \theta^r_t)\right)^2 . \tag{4.1}$$

We can then define a new return that replaces the sampled reward with its expectation:

$$\hat{G}^{\lambda}_{t:t+n} = \hat{v}(s_t; \theta^v_t) + \sum_{k=0}^{n-1} (\gamma\lambda)^k \hat{\delta}_{t+k} \nabla_{\theta^v} \hat{v}(s_t; \theta^v_t), \tag{4.2}$$

where $\hat{\delta}_t = \sum_{a \in \mathcal{A}} \pi(a|s_t)\hat{r}(s_t, a; \theta_t^r) + \gamma\hat{v}(s_{t+1}) - \hat{v}(s_t; \theta_t^v)$ is the one-step TD error with the expected reward replacing the sampled reward. The value function can then be updated using this return:

$$g_{\hat{TD}}(\theta_t^v) = \left(\hat{G}^{\lambda}_{t:t+n} - \hat{v}(s_t; \theta_t^v)\right) \nabla_{\theta^v} \hat{v}(s_t; \theta_t^v). \tag{4.3}$$

We recall that for actor-critic methods, the policy parameters are trained to maximize the return with a baseline used to reduce variance, see section 2.7. Using our reward estimator we have the following policy gradient update:

$$g_{\hat{AC}}(\theta_t^{\pi}) = \nabla_{\theta^{\pi}} \log \pi(s_t, a_t; \theta_t^{\pi})(\hat{G}^{\lambda}_{t:t+n} - \hat{v}(s_t; \theta_t^v)). \tag{4.4}$$

An example of using the reward estimator in an actor-critic process can be seen in Figure 4.1.

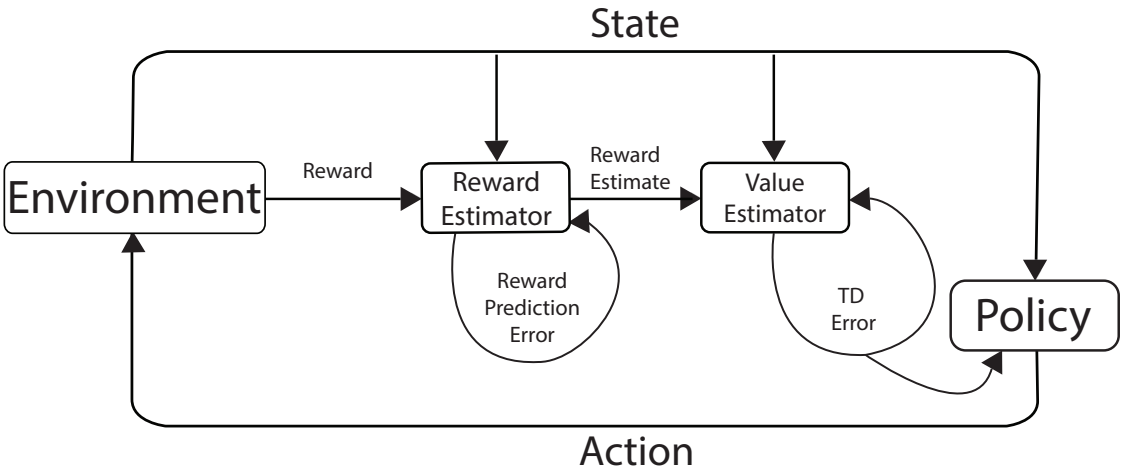

Figure 4.1: The actor-critic update process with the reward estimator. It mostly follows the standard actor-critic update paradigm, except for the important distinction that an estimate of the reward is used to train the value function and the policy.

4.3 Theoretical Variance Reduction

To determine whether using a reward estimator reduces variance theoretically, we examine the tabular case. In this setting, we use the sample mean for the reward: $\hat{r}(s, a) = \frac{1}{N}\sum_i r^i$ with $r^i \sim r(s, a)$. That is, given N independently and identically distributed (i.i.d.) reward samples at a given state s where action a was taken, we determine the mean of those rewards.

First, following a similar methodology to the variance analysis by van Seijen et al. (2009), we determine the variance of the standard one-step Bellman target: $G_{t:t+1} = r_t + \gamma\hat{v}(s_{t+1})$.

The variance of this Bellman target is:

$$\text{var}\left[G_{t:t+1}\right] = \text{var}\left[r_t\right] + \text{var}\left[\gamma\hat{v}(s_{t+1})\right] + 2\,\text{cov}\left[r_t, \gamma\hat{v}(s_{t+1})\right]. \tag{4.5}$$

If we instead use the sample mean of the reward, the target becomes: $\hat{G}_{t:t+1} = \hat{r}(s_t, a_t) + \gamma\hat{v}(s_{t+1})$. Similarly, the variance becomes:

$$\text{var}\left[\hat{G}_{t:t+1}\right] = \text{var}\left[\hat{r}(s_t, a_t)\right] + \text{var}\left[\gamma\hat{v}(s_{t+1})\right] + 2\,\text{cov}\left[\hat{r}(s_t, a_t), \gamma\hat{v}(s_{t+1})\right]. \tag{4.6}$$

Moreover, since we are using the sample mean, we have that:

$$\text{var}\left[\hat{r}(s_t, a_t)\right] = \frac{1}{N}\,\text{var}\left[r_t\right] \tag{4.7}$$

and,

$$\text{cov}\left[\hat{r}(s_t, a_t), \gamma\hat{v}(s_{t+1})\right] = \frac{1}{N}\,\text{cov}\left[r_t, \gamma\hat{v}(s_{t+1})\right] \tag{4.8}$$

Thus, we arrive at the following equality:

$$\text{var}\left[\hat{G}_{t:t+1}\right] - \text{var}\left[G_{t:t+1}\right] = \frac{1-N}{N}\,\text{var}\left[r_t\right] + 2\frac{1-N}{N}\,\text{cov}\left[r_t, \gamma\hat{v}(s_{t+1})\right]. \tag{4.9}$$

We note that with multi-step and λ-returns a similar analysis can be done, with the main difference being that there will be added covariance terms that complicate the interpretation. Analyzing equation (4.9), we see that if the covariance between the reward component and the value function at the next state is ≥ 0 that $\text{var}\left[\hat{G}_{t:t+n}\right] \leq \text{var}\left[G_{t:t+n}\right], \forall N \geq 1$. This may not always be the case, for example, consider an object collection task where the agent is tasked with collecting all of the objects on a particular grid. If we assume that the agent receives a positive reward for collecting an object, then after collection, the value at the next time step may be lower since there is one less object to collect. Thus, the reward and the value at the next time-step may indeed have a negative covariance. In these instances, reward estimation may not reduce the variance of the training target. Nevertheless, we may expect to have success when the variance in the return is largely attributed to stochasticity in the reward function and not in the covariance between the current reward the value at the next state.

4.4 Experiments

To validate that using an estimate for the reward improves performance, we investigate several settings with stochastic noise added to the reward function. First, we use a small toy MDP

problem to empirically validate our theoretical variance reductions shown in section 4.3. We then conduct experiments using several Atari games from the ALE (Bellemare et al., 2013) and MuJoCo tasks from OpenAI Gym (Brockman et al., 2016). The code for all experiments is provided at github.com/facebookresearch/reward-estimator-corl.

4.4.1 Tabular Experiments

We first empirically investigate the tabular case. We construct a 5-state MDP for policy evaluation. The MDP contains deterministic transitions from left to right in all states, and the episode terminates upon reaching the right-most state. At each state, the agent receives a stochastic reward r with a probability of 0.5 and receives a 0 reward with equal probability. The value function is updated via the one-step TD error for 100 episodes. We measure the robustness to variance by evaluating the root mean squared error (RMSE) of the value function over the 100 episodes. As seen in figure 4.2, when using the reward estimator (trained using the sample mean), the agent is able to learn more accurate value functions even at high learning rates.

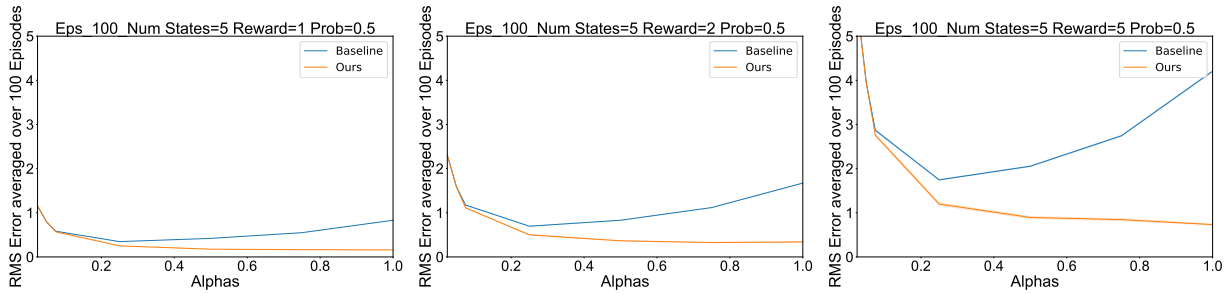

Figure 4.2: Tabular experiments with a 5 state MDP. In all cases, rewards are assigned with probability 0.5 and, set to 0 otherwise (rewards of $+1, +2, +5$, from left to right). The x-axis demonstrates various learning rates for the TD-update. We report the average RMSE over the first 100 episodes of learning - lower is better.

4.4.2 Atari Experiments

Next, we experiment on 5 different tasks from the ALE (Bellemare et al., 2013): Beam Rider, Breakout, Pong, Qbert, Seaquest, and Space Invaders. We build our code off of the A2C implementation from Kostrikov (2018) which uses the exact same hyperparameters as Dhariwal et al. (2017). For the reward estimator, we use an additional network with the same architecture as the value network and use it to train our critic and policy as described in section 4.2. We compare our approach to the standard A2C algorithm, as well as A2C

Environment	$\sigma = 0.0$ (% Gain)	$\sigma = 0.1$ (% Gain)	$\sigma = 0.2$ (% Gain)	$\sigma = 0.3$ (% Gain)	$\sigma = 0.4$ (% Gain)
BeamRider	**26.87**	**49.45**	**1350.95**	**876.43**	**485.13**
Breakout	-1.24	**15.40**	**101.82**	**681.86**	**2152.73**
Pong	-0.22	**21.66**	-1.55	**1882.6**	**32.05**
Qbert	-37.57	-10.18	**78.55**	**456.57**	**646.32**
Seaquest	-29.53	-9.18	-8.68	**74.66**	**115.86**
SpaceInvaders	-10.48	**8.46**	**55.10**	**136.29**	**364.82**
Average	-8.69	**12.6**	**262.7**	**684.73**	**632.82**

Table 4.1: Comparison of the average episode reward over 10M steps of training between our approach to the best of both baselines (A2C and A2C with the reward prediction auxiliary task). The score represents the relative improvement over the best baseline normalized by the performance of the random policy: $\frac{\text{Ours}-\text{Best Baseline}}{|\text{Best Baseline}-\text{Random Policy}|}$. Bold scores indicate an improvement over both baselines. The results are the averaged over 3 runs using different random seeds. Variance of the added Gaussian noise is denoted as σ^2.

with reward prediction as an auxiliary task, which uses reward prediction as an additional loss function to help regularize the network's representation (Jaderberg et al., 2016).

We tuned the learning rate for the reward-predictor through a coarse grid-search ($\alpha \in [0.0001, 0.00025, 0.0005, 0.00075, 0.001]$) on Pong, and then used the best one (0.0001) on all other games. Additionally, we found that occasionally our algorithm diverged completely due to poor initialization of the reward estimator. To alleviate this issue, we provided a convex combination between the estimate and the stochastic environment reward, which we linearly decayed over the first 25000 network updates (out of the total 125000 updates). Finally, we model the reward estimate by simply providing the current state as input (and not the action), which we find to be sufficient, since rewards in the games that were tested are often delayed by several time-steps.

The noise which we add is 0-centered *Gaussian* noise. That is, the resulting reward becomes $r_t = r(s_t, a_t) + \epsilon$ where $\epsilon \sim \mathcal{N}(0, \sigma^2)$. This noise is inspired by a reward signal which is sensory based, but where sensors exhibit a Gaussian distribution as in Nguyen et al. (2012). We note that since only 3 seeds were used for this set of experiment, small gains may be attributed to noise in the learning process. However, as can be seen by the results presented in table 4.1, once a certain noise threshold has been reached, reward estimation provides a large boost in performance. We also see that in scenarios with a small amount of noise that the best baseline often performs similarly or even outperforms our approach. We hypothesize that this is caused by the small bias that may be added if the reward estimate is incorrect with its estimate.

4.4.3 MuJoCo Experiments

Next, we experiment on four different continuous control tasks in the MuJoCo simulator provided by OpenAI Gym (Brockman et al., 2016): Reacher, Hopper, HalfCheetah, and Walker2d. We use PPO (Schulman et al., 2017) instead of A2C as our core learning algorithm, as it tends to outperform A2C on continuous control tasks. Our architecture and hyper-parameters are identical to the standard PPO parameters used in Dhariwal et al. (2017), and our implementation is directly taken from Kostrikov (2018).

Similar to the Atari experiments, we compare against two baselines: standard PPO and PPO with the reward auxiliary task added to the value network. We also use a completely separate network to train the reward estimate with the same structure and the same learning rate as the value network. Moreover, we use a convex combination between our estimate $\hat{r}$ and the stochastic environment reward which we linearly decay over the first 100 network updates (out of the total ~ 500 updates). Finally, unlike in the Atari experiments, we model the reward estimate by including both the state and action as an input. We find that this helps in certain tasks, since some provide a reward for the action itself (e.g., Reacher).

Gaussian noise experiment: Identical to section 4.4.2, the first type of reward noise which we add is 0-centered Gaussian noise. The results of this experiment can be found in table 4.4.3. As can be seen under a 0-centered Gaussian noise, reward estimation improves results over the baseline in all cases except when no noise is added.

	$\sigma = 0.0$ (% Gain)	$\sigma = 0.1$ (% Gain)	$\sigma = 0.2$ (% Gain)	$\sigma = 0.3$ (% Gain)	$\sigma = 0.4$ (% Gain)
Hopper	-8.09	**4.05**	**6.15**	**10.39**	**33.42**
Walker	-8.09	**63.67**	**159.03**	**177.59**	**150.60**
Reacher	-1.79	**10.41**	**16.60**	**30.72**	**24.73**
HalfCheetah	-12.55	**38.70**	**115.21**	**139.52**	**493.61**
Average	-7.63	**29.21**	**74.25**	**89.55**	**175.59**

Table 4.2: Gaussian reward noise ($\sigma = (0.0, 0.1, 0.2, 0.3, 0.4)$) comparison between our approach and the best of both baselines (PPO and PPO with the reward prediction auxiliary task). The score represents the relative improvement over the best baseline normalized with respect to the the average episode reward over the last 100 episodes after training for 1M steps: $\frac{\text{Ours}-\text{Best Baseline}}{|\text{Best Baseline}-\text{Random Policy}|}$. Bold scores indicate an improvement over both baselines. The results are the average over 10 runs using different random seeds.

Sparsity experiment: Next, we consider artificially making the reward sparser by replacing the true environmental reward with the 0 reward with varying levels of probability. Specifically, the reward at time t is defined as:

$$r_t = \begin{cases} 0, & \text{with probability } \epsilon \\ r(s_t, a_t), & \text{with probability } (1-\epsilon). \end{cases} \tag{4.10}$$

This may reflect a scenario where there is a signal dropout either in a sensory-based reward signal as in Potter et al. (2010) or in communication of the reward signal from a human to an agent. This type of noise in particular provides insight into the the effectiveness of $\hat{r}$ in the estimation of sparse rewards. We note that sparsity noise can be seen as simply multiplying the reward signal by a constant positive factor between $[0, 1]$ at every time step, which preserves the *optimal ordering* of policies. Specifically, if $Q(s, a) > Q(s, a)$ under $r(s, a)$ then $Q'(s, a) > Q'(s, a)$ under $r'(s, a) * c$ where $c > 0$. Due to the potential learning problems with sparse rewards (Bellemare et al., 2016), we can see that in table 4.3 that the results are no longer monotonic and that small gains may be attributed to noise in the learning process. However, consistent with previous results, once a certain noise threshold has been reached, reward estimation improves performance in all cases.

	$\epsilon = 0.6$ (% Gain)	$\epsilon = 0.7$ (% Gain)	$\epsilon = 0.8$ (% Gain)	$\epsilon = 0.9$ (% Gain)	$\epsilon = 0.95$ (% Gain)
Hopper	**16.31**	-8.0	**2.00**	**72.54**	**81.93**
Walker	**6.19**	**17.54**	**32.18**	**205.12**	**130.63**
Reacher	-9.98	-16.35	-18.32	-34.69	**83.29**
HalfCheetah	-12.4	**14.5**	-0.67	-6.01	**124.15**
Average	**0.03**	**1.92**	**3.81**	**59.24**	**105**

Table 4.3: Sparse reward noise ($\epsilon = (0.6, 0.7, 0.8, 0.9, 0.95)$) comparison between our approach to the best of both baselines (PPO and PPO with the reward prediction auxiliary task). The score represents the relative improvement as in Figure 4.4.3. The results are the averaged over 10 different experiment random seeds.

4.5 Discussion

In this chapter, we presented a simple method for effectively decomposing the Bellman equation by learning an estimate of the reward function to aid the training of an RL system. Theoretically, we showed that in certain tabular settings this results in variance reduction in

the training target for the value function. Empirically, we demonstrated that once a certain noise threshold was reached, reward estimation provided a large boost of performance over standard actor-critic methods.

An early version of the work presented in this chapter (Romoff et al., 2018b), used a shared network to train the value function and the reward function, with the final linear layer mapping to both the value function and the estimated reward. In later work, we found that the performance was further improved once the reward was estimated via its own neural network (Romoff et al., 2018a). We hypothesize that once the reward estimation problem was fully separated from value estimation, the network was able to fit to the large amount of noise that was being injected into the problem.

We recall that in the introduction of this chapter we mentioned that value functions with a smaller γ converge faster asymptotically than those with a larger γ, see section 2.1 for more details. However, there are two caveats to that analysis, 1) it assumes that the value function is updated deterministically and 2) that the learning rate is set to 1. Empirically, we showed that even with both of these assumptions lifted, the reward estimate can still aid the training of value functions. One explanation could be the fact that unlike the value function, the reward function trained with both the state and action as an input, is mostly invariant [1] to changes in the policy. Thus, even as the policy was improving during training, the reward estimate was still useful, providing variance reduction to the TD target.

In the following chapter, we extend this approach by using the estimate of a value function based on a small γ to aid the learning of value functions with larger γ. While this extension does remove the policy invariance previously discussed, we highlight additional theoretical and empirical benefits that makes this new approach a strong tool for value estimation.

[1] The distribution of states may change when the policy changes.

Chapter 5

Time-Scale Decomposition

The following chapter introduces a method for decomposing the Bellman equation based on the discount factor. In the iterative setting, the discount factor is part of the problem formulation and is not treated as a hyperparameter to be tuned. However, in the RL setting, discounting is often used as a biased proxy for optimizing the cumulative reward to make learning more efficient and stable (Bertsekas and Tsitsiklis, 1995; Prokhorov and Wunsch, 1997; Even-Dar and Mansour, 2003). The optimal discount factor, which balances asymptotic policy performance with learning ability, is often difficult to choose, and solutions have ranged from scheduled curricula (Prokhorov and Wunsch, 1997; François-Lavet et al., 2015; OpenAI, 2018) to meta-gradient learning of the discount factor (Xu et al., 2018).

OpenAI (2018) and François-Lavet et al. (2015), for example, start with a small discount factor and gradually increase it to bootstrap the learning process. Rather than explicitly tackling the problem of discount selection, we make the observation that for any discount factor, the discounted value function already encompasses all smaller time-scales. This simple observation allows us to derive a novel method of generating separable value functions. We can separate the value function into a number of partial estimators, which we call *delta* estimators, that approximate the difference between value functions at different discount factors. More importantly, each delta estimator satisfies a Bellman-like equation based on the value functions of shorter horizons. Thus, these delta estimators can then be summed to yield the same discounted value function, and any subset of estimators from the series of smaller γ_z values. Learning the difference (delta) between value functions leads us to call our method TD(Δ).

The separable nature of the TD(Δ) estimator allows for each component to be learned in a way that is optimal for that part of the overall value function. This means that, for example, the learning rate and the n-step or λ-return can be adjusted for each component, yielding overall faster convergence. Additionally, we provide an intuitive method for setting

intermediary γ_z values which yields performance gains, in most cases, without additional tuning. We demonstrate these benefits theoretically using a similar bias-variance analysis as Kearns and Singh (2000), by adjusting the n-step returns to update each delta estimator. We also show how this method can be combined with TD(λ) (Sutton, 1988) and PPO (Schulman et al., 2017) leading to empirical gains in dense reward Atari games.

5.1 Related work

There are several RL works that learn an ensemble of value functions at different time-scales. Specifically, Feinberg and Shwartz (1994) examine learning an optimal policy for the mixture of two value functions with different discount factors. Similarly, Reinke et al. (2017) learn q-values for multiple discount factors in order to approximate the average return. Sherstan et al. (2018) train a value function such that it can be queried for a given set of time-scales. Finally, concurrent to this work, Fedus et al. (2019) re-weight multiple value functions across different discount factors to form a hyperbolic value function. However, we note that none of the aforementioned works utilize short term estimates to train the longer term value functions. While our method can similarly be used to query smaller time-scales, this is a side-benefit to the performance increases yielded by separating value functions into different time-scales via TD(Δ).

Some recent work has investigated how to precisely select the discount factor (François-Lavet et al., 2015; Xu et al., 2018). François-Lavet et al. (2015) suggest a particular scheduling mechanism, seen similarly in OpenAI (2018) and Prokhorov and Wunsch (1997). Xu et al. (2018) propose a meta-gradient approach which learns the discount factor (and λ value) over time. All of these methods can be applied to our own as we do not necessarily prescribe a final overall γ value to be used.

Finally, hierarchical reinforcement learning methods often decompose value functions or reward functions into a number of smaller systems which can be optimized separately (Dietterich, 2000; Hengst, 2002; Menache et al., 2002; Russell and Zimdars, 2003; van Seijen et al., 2017). These works learn hierarchical policies, paired with the decomposed value functions, which reflect the structure of the goals. Notably, some hierarchical methods, often also require some form of domain knowledge. For example, in our reward decomposition from chapter 3, we require that the reward function be decomposed into a meaningful way before training, which then can be efficiently learned by separate value functions. The approach presented in this chapter, on the other hand, can be applied to any MDP, without any additional prior knowledge, making it a more general solution.

5.2 Proposed Approach

In this section, we introduce TD(Δ), along with several variations, including: multi-step TD and TD(λ). We also show how TD(Δ) can be used for control in policy gradient methods.

5.2.1 Single-step TD(Δ)

Consider learning with $Z+1$ different discount factors $\Delta := \gamma_0, \gamma_1, \ldots, \gamma_Z$. Each of these define a corresponding value function v_z. We define the *delta functions* w_z by

$$w_z := v_z - v_{z-1}, \qquad w_0 := v_0. \tag{5.1}$$

This results in $Z+1$ delta functions such that the desired v_z is simply the sum of the delta functions:

$$v_z(s) = \sum_{i=0}^{z} w_i(s). \tag{5.2}$$

We can derive a Bellman-like equation for the delta functions. Indeed, $w_0 = v_0$ satisfies the Bellman equation:

$$w_0(s_t) = \mathbb{E}^\pi \left[r_t + \gamma_0 w_0(s_{t+1}) \right], \tag{5.3}$$

while the delta functions at larger time-scales satisfy:

$$\begin{aligned} w_z(s_t) &= v_z(s_t) - v_{z-1}(s_t) \\ &= \mathbb{E}^\pi \left[(r_t + \gamma_z v_z(s_{t+1})) - (r_t + \gamma_{z-1} v_{z-1}(s_{t+1})) \right] \\ &= \mathbb{E}^\pi \left[\gamma_z \left(w_z(s_{t+1}) + v_{z-1}(s_{t+1}) \right) - \gamma_{z-1} v_{z-1}(s_{t+1}) \right] \\ &= \mathbb{E}^\pi \left[(\gamma_z - \gamma_{z-1}) v_{z-1}(s_{t+1}) + \gamma_z w_z(s_{t+1}) \right]. \end{aligned} \tag{5.4}$$

This is a Bellman-like equation for w_z, with decay factor γ_z and rewards $(\gamma_z - \gamma_{z-1}) v_{z-1}(s_{t+1})$. Thus, we can use it to define the expected TD update for w_z. Note that in this expression, $v_{z-1}(s_{t+1})$ can be expanded as the sum of $w_i(s_{t+1})$ for $i \leq z-1$, so that the Bellman equation for w_z depends on the values of all delta functions w_i for $i \leq z-1$. Specifically, we use the following one-step return:

$$G^z_{t:t+1} := (\gamma_z - \gamma_{z-1}) \sum_{u=0}^{z-1} \hat{w}_u(s_{t+1}) + \gamma_z \hat{w}_z(s_{t+1}) \tag{5.5}$$

Therefore, the delta value function at a given time-scale appears as an autonomous reinforcement learning problem with rewards coming from the value function of the immediately lower

time-scale. Moreover, we can train all the delta components in parallel according to this TD update, bootstrapping off of the old value of all the estimators. This requires that a sequence of γ_z values is defined beforehand, including a largest and smallest discount γ_0 and γ_Z. We will see in section 5.4.1 that these settings can affect results. However, to avoid the addition of a number of hyperparameters, we assume a simple sequence where we double the *effective planning horizon* (Kearns and Singh, 2002) of the γ_z values, i.e.,

$$\frac{1}{1-\gamma_z} = 2 * \frac{1}{1-\gamma_{z-1}}, \tag{5.6}$$

until the final γ_Z value is reached. This simple sequence of discount factors, without tuning, yields performance gains in many settings as seen in section 5.4.

5.2.2 Multi-step TD(Δ)

In many scenarios, it has been shown that multi-step TD is more efficient than single-step TD (Sutton and Barto, 2018). We can easily extend TD(Δ) to the multi-step case as follows. To begin, since $w_0 := v_0$, the multi-step target for w_0 is identical to the standard multi-step target with $\gamma = \gamma_0$:

$$w_0(s_t) = \mathbb{E}^\pi \left[\sum_{k=0}^{n_0-1} \gamma_0^i r_{t+k} + \gamma_0^{n_0} w_0(s_{t+n_0}) \right]. \tag{5.7}$$

For all other w functions, we can unroll both the bootstrap term and the rewards from the previous value function:

$$\begin{aligned}
&w_z(s_t) = \mathbb{E}^\pi \left[(\gamma_z - \gamma_{z-1}) v_{z-1}(s_{t+1}) + \gamma_z w_z(s_{t+1}) \right] \\
&= \mathbb{E}^\pi \left[(\gamma_z - \gamma_{z-1}) r_{t+1} + \gamma_{z-1}(\gamma_z - \gamma_{z-1}) v_{z-1}(s_{t+2}) + \gamma_z(\gamma_z - \gamma_{z-1}) v_{z-1}(s_{t+2}) + \gamma_z^2 w_z(s_{t+2}) \right] \\
&= \mathbb{E}^\pi \left[(\gamma_z - \gamma_{z-1}) r_{t+1} + (\gamma_z^2 - \gamma_{z-1}^2) v_{z-1}(s_{t+2}) + \gamma_z^2 w_z(s_{t+2}) \right] \\
&= \mathbb{E}^\pi \left[\sum_{k=1}^{n_z-1} (\gamma_z^k - \gamma_{z-1}^k) r_{t+k} + (\gamma_z^{n_z} - \gamma_{z-1}^{n_z}) v_{z-1}(s_{t+n_z}) + \gamma_z^{n_z} w_z(s_{t+n_z}) \right].
\end{aligned} \tag{5.8}$$

Finally, sampling from this expectation and using the sum of the delta estimators in place of the value function gives us the following return:

$$G^z_{t:t+n_z} := \sum_{k=1}^{n_z-1} (\gamma_z^k - \gamma_{z-1}^k) r_{t+k} + (\gamma_z^{n_z} - \gamma_{z-1}^{n_z}) \sum_{u=0}^{z-1} \hat{w}_u(s_{t+n_z}) + \gamma_z^{n_z} \hat{w}_z(s_{t+n_z}).$$

Each w_z receives a fraction of the rewards from the environment up to time-step $n_z - 1$. Additionally, each w_z bootstraps off of its own value function as well as the value at the

previous time-scale.

We recall that truncated TD(λ) uses the following λ-return as a target for its update rules:

$$G^{\gamma,\lambda}_{t:t+n} = \hat{v}_\gamma(s_t) + \sum_{k=0}^{n-1} (\lambda\gamma)^k \delta^\gamma_{t+k}, \tag{5.9}$$

where we add the superscript γ to highlight the dependence on the discount factor.

Similarly, for each w_z we can define the truncated λ-return as:

$$G^{z,\lambda_z}_{t:t+n_z} := \hat{w}_z(s_t) + \sum_{k=0}^{n_z-1} (\lambda_z\gamma_z)^k \delta^z_{t+k}, \tag{5.10}$$

where $\delta^0_t := \delta^{\gamma_0}_t$ and $\delta^z_t := (\gamma_z - \gamma_{z-1}) \sum_{u=0}^{z-1} \hat{w}_u(s_{t+1}) + \gamma_z \hat{w}_z(s_{t+1}) - \hat{w}_z(s_t)$ are the one-step TD errors.

5.2.3 TD(λ, Δ) with Policy Gradient Methods

Since TD(λ) is used in powerful policy gradient baselines (Schulman et al., 2017), we propose to train the policy and the value function using the following advantage function:

$$A^{\Delta,\lambda}_{t:t+n}(s_t) := \sum_{k=0}^{n-1} (\lambda\gamma)^k \delta^\Delta_{t+k}, \tag{5.11}$$

where $\delta^\Delta_{t+k} := r_t + \gamma_Z \sum_{z=0}^{Z} \hat{w}_z(s_{t+1}) - \sum_{z=0}^{Z} \hat{w}_z(s_t)$. Thus, the sum of all our w estimators are used as a replacement for v. This objective can easily be applied to PPO by using the policy update from equation (2.50) and replacing $A^\lambda_{t:t+n}$ with $A^{\Delta,\lambda}_{t:t+n}$. Similarly, to train each w_z, we use the truncated λ-return defined in equation (5.10).

5.3 Theoretical Analysis

We now analyze our estimators more formally. The goal is that our estimator will provide favorable bias-variance trade-offs under some circumstances. To shed light on this, we start by illustrating when our estimator is identical to the single estimator (theorem 5.3.1). Then motivated by these results and prior work by Kearns and Singh (2000), we bound the error of our estimator in terms of a bias and variance term (theorem 5.3.4).

5.3.1 Equivalence settings and improvement

In some cases, we can show that our TD(Δ) update and its variations are equivalent to the non-delta estimator v_γ when recomposed into a value function. In particular, we focus here on linear function approximation of the form:

$$\hat{v}_\gamma(s) := \langle \theta^\gamma, x(s) \rangle \quad \text{and} \quad \hat{w}_z(s) := \langle \theta^z, x(s) \rangle, \forall z$$

where θ^γ and $\{\theta^z\}_z$ are weight vectors in $\mathbb{R}^d$ and $x : \mathcal{S} \to \mathbb{R}^d$ is a feature map from a state to a given d-dimensional feature space. Let θ^γ be updated using TD(λ) as follows:

$$\theta^\gamma_{t+1} = \theta^\gamma_t + \alpha \left(G^{\gamma,\lambda}_{t:t+n} - \hat{v}_\gamma(s_t) \right) x(s_t), \tag{5.12}$$

where $G^{\gamma,\lambda}_{t:t+n}$ is the TD(λ) return defined in equation (5.9).

Similarly, each $\hat{w}_z$ is updated using TD(λ_z, Δ) as follows:

$$\theta^z_{t+1} = \theta^z_t + \alpha_z \left(G^{z,\lambda_z}_{t+n_z} - \hat{w}_z(s_t) \right) x(s_t), \tag{5.13}$$

where $G^{z,\lambda_z}_{t:t+n_z}$ is the TD(Δ) return defined in equation (5.10). Here, α and $\{\alpha_z\}_z$ are positive learning rates. The following theorem establishes the equivalence of the two algorithms.

Theorem 5.3.1. *If $\alpha_z = \alpha, \lambda_z \gamma_z = \lambda\gamma, n_z = n, \forall z$ and if we pick the initial conditions such that $\sum_{z=0}^{Z} \theta^z_0 = \theta^\gamma_0$, then the iterates produced by TD(λ) from equation* (5.12) *and TD(λ, Δ) from equation* (5.13) *with linear function approximation satisfy:*

$$\sum_{z=0}^{Z} \theta^z_t = \theta^\gamma_t, \forall t, \tag{5.14}$$

Proof. The proof is by induction. We first need to show that the base case holds at $t = 0$. This is true given the assumption on initialization. We also note that the base case holds with zero-initialization.

Next, we assume that the statement holds for a given time-step t, i.e., $\sum_{z=0}^{Z} \theta^z_t = \theta^\gamma_t$ and show that it holds at next time-step $t + 1$.

$$\sum_{z=0}^{Z} \theta_{t+1}^{z} = \sum_{z=0}^{Z} \left(\theta_t^z + \alpha_z \left(G_{t:t+n_z}^{z,\lambda_z} - \hat{w}_z(s_t) \right) x(s_t) \right) \tag{5.15}$$

$$= \theta_t^{\gamma} + \sum_{z=0}^{Z} \alpha_z \left(\sum_{k=0}^{n_z-1} (\lambda_z \gamma_z)^k \delta_{t+k}^z \right) x(s_t) \qquad \text{(by induction)} \tag{5.16}$$

$$= \theta_t^{\gamma} + \alpha \sum_{k=0}^{n-1} (\lambda\gamma)^k \underbrace{\left(\sum_{z=0}^{Z} \delta_{t+k}^z \right)}_{(\star)} x(s_t) \qquad \text{(since } \alpha_z = \alpha, \lambda_z\gamma_z = \lambda\gamma, n_z = n, \forall z\text{)} \tag{5.17}$$

Thus, if $(\star) = \sum_{z=0}^{Z} \delta_t^z = \delta_t^{\gamma}$ for any t, then we will have an identical update and $\sum_{z=0}^{Z} \theta_{t+1}^z = \theta_{t+1}^{\gamma}$.

$$\begin{aligned}
\sum_{z=0}^{Z} \delta_t^z &= r_t + \gamma_0 \hat{w}_0(s_{t+1}) - \hat{w}_0(s_t) + \sum_{z=1}^{Z} \left((\gamma_z - \gamma_{z-1}) \sum_{u=0}^{z-1} \hat{w}_u(s_{t+1}) + \gamma_z w_z(s_{t+1}) - \hat{w}_z(s_t) \right) \\
&= r_t + \sum_{u=0}^{Z} \sum_{z=u+1}^{Z} (\gamma_z - \gamma_{z-1}) \hat{w}_u(s_{t+1}) + \sum_{z=0}^{Z} \gamma_z \hat{w}_z(s_{t+1}) - \sum_{z=0}^{Z} \hat{w}_z(s_t) \\
&= r_t + \sum_{u=0}^{Z} (\gamma_Z - \gamma_u) \hat{w}_u(s_{t+1}) + \sum_{z=0}^{Z} \gamma_z \hat{w}_z(s_{t+1}) - \sum_{z=0}^{Z} \hat{w}_z(s_t) \\
&= r_t + \gamma_Z \sum_{z=0}^{Z} \hat{w}_z(s_{t+1}) - \sum_{z=0}^{Z} \hat{w}_z(s_t) = r_t + \gamma \hat{v}_{\gamma}(s_{t+1}) - \hat{v}_{\gamma}(s_t) = \delta_t^{\gamma}
\end{aligned} \tag{5.18}$$

□

Similarly, we can consider learning each w_z using n-step TD(Δ) instead of TD(λ, Δ). In this case, the analysis of theorem 5.3.1 immediately applies since with $\lambda = 1$ the n-step truncated λ return is equivalent to the n-step return. Note that the equivalence is achieved when $\lambda_z\gamma_z = \lambda\gamma, \forall z$. When λ is close to 1 and $\gamma_z < \gamma$, the latter condition implies that $\lambda_z = \frac{\lambda\gamma}{\gamma_z}$ could potentially be larger than one. One would conclude that the TD(λ_z) could diverge. Fortunately, we show in the next theorem that the TD(λ) operator defined in equation (2.22) is a contraction mapping for $1 \leq \lambda < \frac{1+\gamma}{2\gamma}$ which implies that $\lambda\gamma < 1$.

Theorem 5.3.2. *$\forall \lambda \in [0, \frac{1+\gamma}{2\gamma}[$, the operator $\mathcal{T}_\lambda v$ defined as $\mathcal{T}_\lambda v = v + (I - \lambda\gamma P)^{-1}(r + \gamma P v - v), \forall v \in \mathbb{R}^{|\mathcal{S}|}$ is well defined. Moreover, $\mathcal{T}_\lambda v$ is a contraction with respect to the max norm and its contraction coefficient is equal to $\frac{\gamma|1-\lambda|}{1-\lambda\gamma}$*

Proof. We recall that from equation (2.24) that the contraction properties of $\mathcal{T}_\lambda$ are determined by:

$$\|\mathcal{T}_\lambda v_1 - \mathcal{T}_\lambda v_2\| \leq \frac{\gamma|1-\lambda|}{1-\lambda\gamma}\|v_1 - v_2\| \tag{5.19}$$

We know that $0 \leq \lambda \leq 1$ is a contraction, for $\lambda > 1$ we need:

$$\begin{aligned} \frac{\gamma(\lambda-1)}{1-\lambda\gamma} < 1 &\Rightarrow \gamma\lambda - \gamma < 1 - \lambda\gamma \\ &\Rightarrow 2\gamma\lambda < 1 + \gamma \Rightarrow \lambda < \frac{1+\gamma}{2\gamma} \end{aligned} \tag{5.20}$$

Therefore, $\mathcal{T}_\lambda$ is a contraction mapping if $0 \leq \lambda < \frac{1+\gamma}{2\gamma}$.

□

However, we note that the equivalence with unmodified TD learning is the exception rather than the rule. For one, in order to achieve equivalence we require the same learning rate across every time-scale. This is a strong restriction as intuitively the shorter time-scales can be learned faster than the longer ones. Further, adaptive optimizers are typically used in the nonlinear approximation setting (Henderson et al., 2018; Schulman et al., 2017). Thus, the effective rate of learning can differ depending on the properties of each delta estimator and its target. In principle, the optimizer can automatically adapt the learning to be different for the shorter and longer time-scales.

Besides for the learning rate, such a decomposition allows for some particularly helpful properties not afforded to the non-delta estimator. In particular, every w_z delta component need not use the same n-step return (or λ-return) as the non-delta estimator (or the higher w_z components). Specifically, if $n_z < n_{z+1}, \forall z$ (or $\gamma_z\lambda_z < \gamma_{z+1}\lambda_{z+1}, \forall z$), then there is the possibility for variance reduction at the risk of some bias introduction.

5.3.2 Analysis for reducing n-step values

To see how our method differs from the single estimator case, let us consider the tabular *phased* version of n-step TD studied by Kearns and Singh (2000). In this setting, starting from each state $s \in \mathcal{S}$, we generate m trajectories:

$$\left\{s_0^{(j)} = s, a_0, r_0, \ldots, s_n^{(j)}, a_n^{(j)}, r_n^{(j)}, s_{n+1}^{(j)}, \ldots\right\}_{1 \leq j \leq m},$$

following policy π. For each iteration t, called also phase t, the value function estimate for s is defined as follows:

$$\hat{v}_{\gamma,t}(s) = \frac{1}{m}\sum_{j=1}^{m}\left(\sum_{k=0}^{n-1}\gamma^k r_k^{(j)} + \gamma^n \hat{v}_{\gamma,t-1}(s_n^{(j)})\right) \tag{5.21}$$

The following theorem from Kearns and Singh (2000) provides an upper bound on the error in the value function estimates defined by $\Delta_t^\gamma := \max_s\{|\hat{v}_{\gamma,t}(s) - v_\gamma(s)|\}$.

Theorem 5.3.3. *(Kearns and Singh, 2000) for any $0 < \delta < 1$, let $\epsilon = \sqrt{\frac{2\log(2n/\delta)}{m}}$. with probability $1-\delta$,*

$$\Delta_t^\gamma \leq \underbrace{\epsilon\left(\frac{1-\gamma^n}{1-\gamma}\right)}_{\text{variance term}} + \underbrace{\gamma^n \Delta_{t-1}^\gamma}_{\text{bias term}}, \tag{5.22}$$

Proof. Hoeffding's inequality guarantees for a variable that is bounded between $[-1,+1]$, that:

$$P\left(\left|\frac{1}{m}\sum_{j=1}^{m} r_k^{(j)} - \mathbb{E}[r_k]\right| \geq \epsilon\right) \leq 2\exp\left(\frac{-2m^2\epsilon^2}{m2^2}\right) \tag{5.23}$$

If we assume that the probability is fixed to be no more than δ, we can solve for the resulting value of ϵ:

$$2\exp\left(\frac{-m^2\epsilon^2}{2m}\right) = \delta \Longrightarrow \frac{-m\epsilon^2}{2} = \log(\delta/2) \Longrightarrow \epsilon = \sqrt{\frac{2\log(2/\delta)}{m}} \tag{5.24}$$

So by Hoeffding's inequality, if we have m samples, with probability at least $1-\delta$,

$$\left|\frac{1}{m}\sum_{j} r_k^{(j)} - \mathbb{E}[r_k]\right| \leq \epsilon = \sqrt{\frac{2\log(2/\delta)}{m}}. \tag{5.25}$$

We want each of the $\mathbb{E}[r_k]$ reward terms in the n-step return to all be estimated up to ϵ accuracy with high probability. To do so, we can use a union bound and assume that the probability that we fail to estimate any of these n expected reward terms is at most δ/n. Substituting this into the above equation for ϵ, we obtain that with probability at least $1-\delta$, each of the $\mathbb{E}[r_k]$ terms are estimated to within $\epsilon = \sqrt{\frac{2\log\frac{2n}{\delta}}{m}}$ accuracy.

Substituting this bound back into the definition of the n-step TD update we get:

$$\begin{aligned}
\hat{v}_{t+1}(s) - v(s) &= \frac{1}{m}\sum_{j=1}^{m}\left(r_0^{(j)} + \gamma r_1^{(j)} + \ldots \gamma^{n-1} r_{n-1}^{(j)} + \gamma^n v_t(s_n^{(j)})\right) - v(s) \\
&= \sum_{k=0}^{n-1}\gamma^k\left(\frac{1}{m}\sum_{j=1}^{m} r_k^{(j)} - \mathbb{E}[r_k]\right) + \gamma^n\left(\frac{1}{m}\sum_{j=1}^{m} v_t(s_n^{(j)}) - \mathbb{E}[v(s_n)]\right) \\
&\leq \sum_{k=0}^{n-1}\gamma^k\epsilon + \gamma^n\left(\frac{1}{m}\sum_{j=1}^{m} v_t(s_n^{(j)}) - \mathbb{E}[v(s_n)]\right) \\
&\leq \epsilon\frac{(1-\gamma^n)}{1-\gamma} + \gamma^n\left(\frac{1}{m}\sum_{j=1}^{m} v_t(s_n^{(j)}) - \mathbb{E}[v(s_n)]\right), \qquad (5.26)
\end{aligned}$$

where the second term is bounded by Δ_{t-1}^{γ} by assumption. Finally, we obtain:

$$\Delta_t^{\gamma} \leq \epsilon\frac{(1-\gamma^n)}{1-\gamma} + \gamma^n\Delta_{t-1}^{\gamma} \qquad (5.27)$$

□

The first term $\epsilon(\frac{1-\gamma^n}{1-\gamma})$ in the bound is a variance term arising from sampling transitions. In particular, ϵ bounds the deviation of the empirical average of rewards from the true expected reward. The second term is a bias term due to bootstrapping off of the current value estimate.

Similarly, we consider a phased version of multi-step TD(Δ). For each phase t, we update each w as follows:

$$\hat{w}_{z,t}(s) = \frac{1}{m}\sum_{j=1}^{m}\left(\sum_{k=1}^{n_z-1}(\gamma_z^k - \gamma_{z-1}^k)r_k^{(j)} + (\gamma_z^{n_z} - \gamma_{z-1}^{n_z})v_{z-1}(s_{n_z}^{(j)}) + \gamma_z^{n_z}\hat{w}_z(s_{n_z}^{(j)})\right). \qquad (5.28)$$

We now establish an upper bound on the error of phased TD(Δ) defined as the sum of error incurred by each w component, $\sum_{z=0}^{Z}\Delta_t^z$, where $\Delta_t^z = \max_s\{|\hat{w}_z(s) - w_z(s)|\}$

Theorem 5.3.4. *Assume that $\gamma_0 \leq \gamma_1 \leq \ldots \gamma_Z = \gamma$ and $n_0 \leq n_1 \ldots \leq n_Z = n$, for any $0 < \delta < 1$, let $\epsilon = \sqrt{\frac{2\log(2n/\delta)}{m}}$, with probability $1-\delta$,*

$$\sum_{z=0}^{Z}\Delta_t^z \leq \epsilon\frac{1-\gamma^n}{1-\gamma} + \underbrace{\epsilon\sum_{z=0}^{Z-1}\frac{\gamma_z^{n_{z+1}} - \gamma_z^{n_z}}{1-\gamma_z}}_{\textit{variance reduction}} + \underbrace{\sum_{z=0}^{Z-1}(\gamma_z^{n_z} - \gamma_z^{n_{z+1}})\sum_{u=0}^{z}\Delta_{t-1}^u}_{\textit{bias introduction}} + \gamma^n\sum_{z=0}^{Z}\Delta_{t-1}^z, \qquad (5.29)$$

Proof. Phased TD(Δ) update rules for $z \geq 1$:

$$\hat{w}_{z,t}(s) = \frac{1}{m}\sum_{j=1}^{m}\left(\sum_{k=1}^{n_z-1}(\gamma_z^k - \gamma_{z-1}^k)r_k^{(j)} + (\gamma_z^{n_z} - \gamma_{z-1}^{n_z})\hat{v}_{\gamma_{z-1}}(s_{n_z}^{(j)}) + \gamma_z^{n_z}\hat{w}_z(s_{n_z}^{(j)})\right) \tag{5.30}$$

We know that according to the multi-step Bellman expectation from equation (5.8) for $z \geq 1$:

$$w_z(s) = \mathbb{E}\left[\sum_{k=1}^{n_z-1}(\gamma_z^k - \gamma_{z-1}^k)r_k + (\gamma_z^{n_z} - \gamma_{z-1}^{n_z})v_{\gamma_{z-1}}(s_{n_z}) + \gamma_z^{n_z}w_z(s_{n_z})\right] \tag{5.31}$$

Then, subtracting the two expressions gives for $z \geq 1$:

$$\begin{aligned}\hat{w}_{z,t}(s) - w_z(s) &= \sum_{k=1}^{n_z-1}(\gamma_z^k - \gamma_{z-1}^k)\left(\frac{1}{m}\sum_{j=1}^{m} r_k^{(j)} - \mathbb{E}[r_k]\right) \\ &+ (\gamma_z^{n_z} - \gamma_{z-1}^{n_z})\left(\sum_{u=0}^{z-1}\hat{w}_u(s_k^{(j)}) - \mathbb{E}[w_z(s_n)]\right) + \gamma_z^{n_z}\left(w_z(s_k^{(j)}) - \mathbb{E}[w_z(s_n)]\right)\end{aligned} \tag{5.32}$$

Assuming that $n_0 \leq n_1 \leq \ldots n_Z = n$, the w estimates have at most $n_Z = n$ reward terms $\frac{1}{m}\sum_{j=1}^{m} r_k^{(j)}$. Thus, using Hoeffding's inequality and the union bound, we obtain that with probability $1-\delta$, each of the n empirical average rewards, $\frac{1}{m}\sum_{j=1}^{m} r_k^{(j)}$, deviates from the true expected reward $\mathbb{E}[r_i]$ by at most $\epsilon = \sqrt{\frac{2\log(2n/\delta)}{m}}$. Hence, with probability $1-\delta$, $\forall z \geq 1$, we have:

$$\begin{aligned}\Delta_t^z &\leq \epsilon\sum_{k=1}^{n_z-1}(\gamma_z^k - \gamma_{z-1}^k) + (\gamma_z^{n_z} - \gamma_{z-1}^{n_z})\sum_{u=0}^{z-1}\Delta_{t-1}^u + \gamma_z^{n_z}\Delta_{t-1}^z \\ &= \epsilon\left(\frac{1-\gamma_z^{n_z}}{1-\gamma_z} - \frac{1-\gamma_{z-1}^{n_z}}{1-\gamma_{z-1}}\right) + (\gamma_z^{n_z} - \gamma_{z-1}^{n_z})\sum_{u=0}^{z-1}\Delta_{t-1}^u + \gamma_z^{n_z}\Delta_{t-1}^z\end{aligned} \tag{5.33}$$

and

$$\Delta_t^0 \leq \epsilon\frac{1-\gamma_0^{n_0}}{1-\gamma_0} + \gamma_0^{n_0}\Delta_{t-1}^0 \tag{5.34}$$

Summing the two previous inequalities gives:

$$\begin{aligned}\sum_{z=0}^{Z}\Delta_t^z &\leq \epsilon\frac{1-\gamma_0^{n_0}}{1-\gamma_0} + \epsilon\sum_{z=1}^{Z}\left(\frac{1-\gamma_z^{n_z}}{1-\gamma_z} - \frac{1-\gamma_{z-1}^{n_z}}{1-\gamma_{z-1}}\right) + \sum_{z=1}^{Z}(\gamma_z^{n_z} - \gamma_{z-1}^{n_z})\sum_{u=0}^{z-1}\Delta_{t-1}^u + \gamma_z^{n_z}\Delta_{t-1}^z \\ &= \underbrace{\epsilon\frac{1-\gamma_Z^{n_Z}}{1-\gamma_Z} + \epsilon\sum_{z=0}^{Z-1}\frac{\gamma_z^{n_{z+1}} - \gamma_z^{n_z}}{1-\gamma_z}}_{(\star)\text{variance term}} + \underbrace{\sum_{z=1}^{Z}(\gamma_z^{n_z} - \gamma_{z-1}^{n_z})\sum_{u=0}^{z-1}\Delta_{t-1}^u + \gamma_z^{n_z}\Delta_{t-1}^z}_{(\star\star)\text{bias term}}\end{aligned} \tag{5.35}$$

Let's focus now further on the bias term ($\star\star$):

$$\begin{aligned}
\sum_{z=1}^{Z}(\gamma_z^{n_z}-\gamma_{z-1}^{n_z})\sum_{u=0}^{z-1}\Delta_{t-1}^u+\gamma_z^{n_z}\Delta_{t-1}^z &= \sum_{u=0}^{Z-1}\sum_{z=u+1}^{Z}(\gamma_z^{n_z}-\gamma_{z-1}^{n_z})\Delta_{t-1}^u+\sum_{z=1}^{Z}\gamma_z^{n_z}\Delta_{t-1}^z \\
&= \sum_{u=0}^{Z-1}\Delta_{t-1}^u\left(\sum_{z=u+1}^{Z}\gamma_z^{n_z}-\sum_{z=u}^{Z-1}\gamma_z^{n_{z+1}}\right)+\sum_{z=1}^{Z}\gamma_z^{n_z}\Delta_{t-1}^z \\
&= \sum_{u=0}^{Z-1}\Delta_{t-1}^u\left(\sum_{z=u+1}^{Z-1}(\gamma_z^{n_z}-\gamma_z^{n_{z+1}})+\gamma_Z^{n_Z}-\gamma_u^{n_{u+1}}\right)+\sum_{z=1}^{Z}\gamma_z^{n_z}\Delta_{t-1}^z \\
&= \sum_{u=0}^{Z-1}\sum_{z=u+1}^{Z-1}(\gamma_z^{n_z}-\gamma_z^{n_{z+1}})\Delta_{t-1}^u+\gamma_Z^{n_Z}\sum_{z=0}^{Z}\Delta_{t-1}^z+\sum_{z=0}^{Z-1}(\gamma_z^{n_z}-\gamma_z^{n_{z+1}})\Delta_{t-1}^z \\
&= \sum_{u=0}^{Z-1}\sum_{z=u}^{Z-1}(\gamma_z^{n_z}-\gamma_z^{n_{z+1}})\Delta_{t-1}^u+\gamma_Z^{n_Z}\sum_{z=0}^{Z}\Delta_{t-1}^z \\
&= \sum_{z=0}^{Z-1}(\gamma_z^{n_z}-\gamma_z^{n_{z+1}})\sum_{u=0}^{z}\Delta_{t-1}^u+\gamma_Z^{n_Z}\sum_{z=0}^{Z}\Delta_{t-1}^z \qquad (5.36)
\end{aligned}$$

Finally, we obtain:

$$\sum_{z=0}^{Z}\Delta_t^z \le \epsilon\frac{1-\gamma^n}{1-\gamma}+\underbrace{\epsilon\sum_{z=0}^{Z-1}\frac{\gamma_z^{n_{z+1}}-\gamma_z^{n_z}}{1-\gamma_z}}_{\text{variance reduction}}+\underbrace{\sum_{z=0}^{Z-1}(\gamma_z^{n_z}-\gamma_z^{n_{z+1}})\sum_{u=0}^{z}\Delta_{t-1}^u}_{\text{bias introduction}}+\gamma^n\sum_{z=0}^{Z}\Delta_{t-1}^z. \qquad (5.37)$$

□

Comparing the bound for phased n-step TD in theorem 5.3.3 with the one for phased TD(Δ) in theorem 5.3.4, we see that the latter allows for a variance reduction equal to $\epsilon\sum_{z=0}^{Z-1}\frac{\gamma_z^{n_{z+1}}-\gamma_z^{n_z}}{1-\gamma_z}\le 0$ but it suffers from a potential bias introduction equal to $\sum_{z=0}^{Z-1}(\gamma_z^{n_z}-\gamma_z^{n_{z+1}})\sum_{u=0}^{z}\Delta_{t-1}^u\ge 0$. This is due to the compounding bias from all shorter-horizon estimates. We note that in the case that n_z are all equal we obtain the same upper bound for both algorithms.

It is a well known and an often used result that the expected discounted return over T steps is close to the infinite-horizon discounted expected return after $T\approx\frac{1}{1-\gamma}$ (Kearns and Singh, 2002). Thus, we can conveniently reduce n_z for any γ_z such that $n_z\approx\frac{1}{1-\gamma_z}$ so that we follow this rule. If we have T samples, we can have an excellent bias-variance trade-off on all time-scales $<< T$ by choosing $n_z=\frac{1}{(1-\gamma_z)}$, so that $\gamma_z^{n_z}$ is bound by a constant (since $\gamma_z^{\frac{1}{1-\gamma_z}}\le\frac{1}{e}$) for all z. This provides intuitive ways to set both γ_z and n_z values (as well as all

other parameters) without necessarily searching. We can double the effective horizon at each increasing w_z (to keep a logarithmic number of value functions with respect to the horizon) and similarly adjust all other parameters for estimation.

5.4 Deep RL Experiments

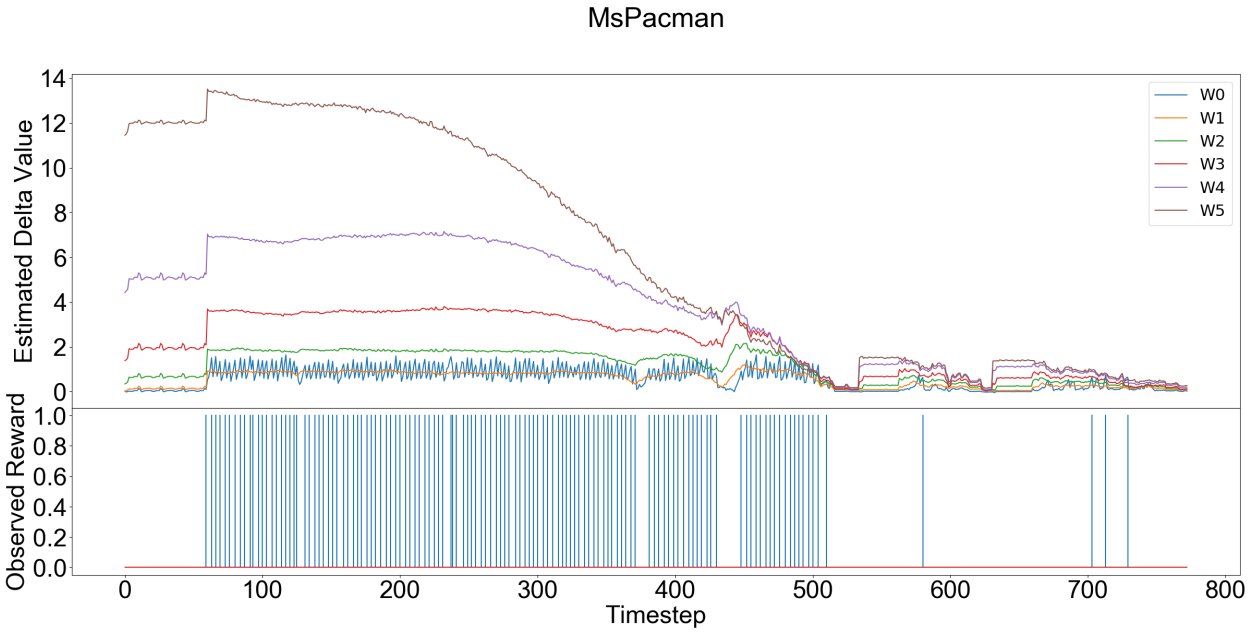

Figure 5.1: The w_z estimators versus the reward over a single episode in Ms. Pac-Man - the drops in value align with a lost life. This is done on a single rollout trajectory of the trained PPO-TD($\hat{\lambda}, \Delta$) agent.

In the following section, we demonstrate performance gains in Atari using the PPO-based version of TD(Δ). We directly update PPO with TD(λ, Δ), using the code of Kostrikov (2018). We compare against PPO as a baseline with standard hyperparameters (Schulman et al., 2017; Kostrikov, 2018). Our architecture differs slightly from the PPO baseline as the value function now outputs $Z + 1$ outputs (1 for each w). We also compare against another neural network architecture which replicates the parameters of TD(Δ). That is, we compare against a value function that outputs $Z + 1$ values which are summed together before computing the value loss (we call this PPO+).

We run two versions of TD(Δ). The first version sets γ_z such that $\gamma_Z = \gamma$ and $\gamma_{z-1} = \frac{1}{2}(\frac{1}{(1-\gamma_z)})$ while $\gamma_z \geq .5$. We then set λ_z for each γ_z such that $\gamma_z \lambda_z = \gamma_Z \lambda_Z$ as per theorem 5.3.1. However, we note that due to the use of an adaptive optimizer (Kingma and Ba, 2014), performance may improve as parameters are automatically adapted for each delta estimator.

Moreover, parity with the baseline model is not necessary and λ can effectively be reduced. To this end, we introduce a second version of our method, labelled PPO-TD($\hat{\lambda}, \Delta$), where we limit $\lambda_z \leq 1$. We hypothesize that values greater than 1 add unnecessary variance to the value estimation problem since $\lambda = 1$ corresponds with the Monte Carlo return.

We run experiments on the 9 games defined in Bellemare et al. (2016) as *hard* with *dense* rewards. We chose hard games as they are most likely to need algorithmic improvements to solve. We omitted *sparse* reward tasks since we do not address the problem of exploration needed for tackling sparse reward settings. Instead, we focus on *dense* reward tasks that may result in complex value functions. As seen in table 5.1, PPO-TD(λ, Δ) performs (statistically) significantly better in a certain class of games roughly related to the frequency of non-zero rewards (the density). Both versions of TD(Δ) perform worse asymptotically than the baselines in two games, Zaxxon and Wizard of Wor, which belong to a class of games with lower density. Though PPO-TD($\hat{\lambda}, \Delta$) performs better in both cases, as we will see in section 5.4.1, it is still possible to improve performance further in these games by tuning the number and scale of γ_Z factors.

One may wonder why performance improves in increasingly dense reward settings. As seen in figure 5.1, it may be due to the separated estimators being able to model fine-grained phenomena. Our hypothesis is that TD(Δ) allows for quick learning of short-term phenomena, followed by slower learning of long-term dependencies. Notice how the long-term w_Z value declines early according to a consistent gradient towards a lost life in the game, while short-term phenomena continue to be captured in the smaller components like w_0.

5.4.1 Tuning and Ablation

In the previous section, we demonstrated how using a fixed set of γ, λ could yield performance gains in a number of environments over the single estimator case. However, a performance drop was seen in the case of Zaxxon and Wizard Of Wor. Due to our bias-variance trade-off in bootstrapping from smaller delta estimators, a curriculum based on smaller horizons may effectively slow learning in some cases. However, the benefit of separating value functions in a flexible way is that they can be tuned. In figure 5.2, we show how different γ values can be used to improve asymptotic performance to match the baseline. For example, by increasing the lowest effective horizon (γ_0) of w_0, we bias the algorithm less toward myopic settings and increase the learning speed to be comparable to the baselines. We note that further tuning of the number of components and their parameters (γ_z, λ_z, α) may further improve performance.

Game	PPO-TD(λ, Δ)	PPO-TD$\left(\hat{\lambda}, \Delta\right)$	PPO+	PPO	Reward Density
Zaxxon	396 ± 210	3291 ± 812	7006 ± 211 †	7366 ± 223 †	1.15
WizardOfWor	2118 ± 138	2440 ± 89	2870 ± 218 †	3408 ± 193 †	1.07
Qbert	13428 ± 333 †	13092 ± 430 †	10594 ± 335	11735 ± 387	12.26
MsPacman	2273 ± 67 †	2241 ± 78 †	1876 ± 89	1888 ± 111	13.27
Hero	29074 ± 512 †	29014 ± 764 †	23511 ± 843	21038 ± 972	13.46
Frostbite	292 ± 7	304 ± 21	299 ± 2	294 ± 5	5.04
BankHeist	1183 ± 13	1166 ± 5	1199 ± 5	1190 ± 3	6.3
Amidar	731 ± 30 †	672 ± 45	611 ± 34	575 ± 54	4.63
Alien	1606 ± 112*	1663 ± 113*	1374 ± 85	1315 ± 70	11.3

Table 5.1: Asymptotic Atari performance (across last 100 episodes) with the mean across 10 seeds and the standard error. † denotes significantly better results over our algorithm in the case of baselines or over the best baseline in the case of our algorithm using Welch's t-test with a significance level of .05 and bootstrap confidence intervals (Colas et al., 2018; Henderson et al., 2017). * indicates significant using bootstrap CI, but not t-test. Bold games are where we perform as well as or significantly better than the baselines. Reward Density is the frequency of rewards per 100 time-steps averaged over $10k$ time-steps under learned policy using baseline (PPO). Notice how the task *Zaxxon* has a much lower frequency than the largest frequency task *Hero*.

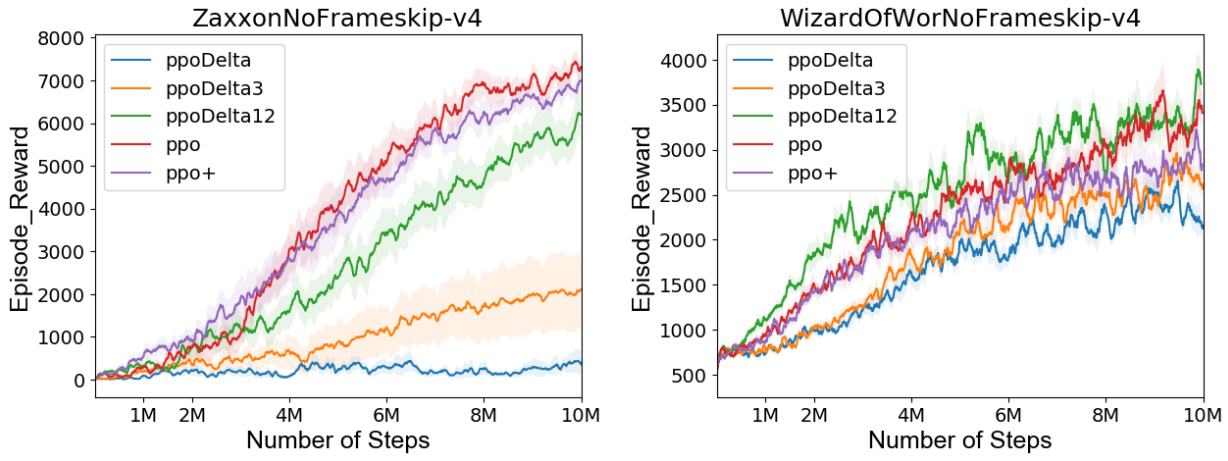

Figure 5.2: Performance of TD(Δ) variations vs. the baselines on Zaxxon and Wizard Of Wor. ppo+ refers to ppo with an augmented architecture. ppoDelta refers to setting $\gamma_z \lambda_z = \gamma\lambda \; \forall z$. ppoDelta3 and ppoDelta12 only use two value functions with horizons $(3, 100)$ and $(12, 100)$ respectively. Shaded regions represent the standard error across 10 random seeds.

5.5 Discussion

In this chapter, we proposed a novel way for decomposing the Bellman equation based on the difference between two value functions with different discount factors. This has convenient theoretical and practical properties which help improve performance in certain settings. These properties have additional benefits: they allow for a natural way to distribute and parallelize training, easy inspection of performance at different discount factors, and the possibility of lifelong learning by adding or removing components. Moreover, we have also highlighted the limitations of this method (introduced bias toward myopic returns) when using the simple parameter settings we proposed. However, these limitations can be overcome with the additional ability to tune parameters at different time-scales. We briefly discuss the added benefits of TD(Δ) below.

While we have not pursued it experimentally here, another benefit of separating value functions in this way is that this reflects a natural way of distributing updates across systems for large scale problems. In fact, prior work has sought different ways to scale RL algorithms through partitioning methods (though typically through other means like dividing the state space) (Wingate and Seppi, 2003; Wingate, 2004). Our work provides another such method for scaling RL systems in a different way. A TD(Δ) update can be spread across many machines, such that each w_z is updated separately (as long as weights are synced across machines after a parallel update).

Many of the performance improvements seen here come not necessarily from the decomposition method itself, but from the ability to set certain parameters differently for each component. The decomposition of the value function allows for further improvement by tuning the number of delta estimators and the γ_z values which correlate with them. In the future, a meta-gradient method as Xu et al. (2018) proposed could be used to automatically scale delta estimators to time-scales which require more computational complexity. However, we note that the default method for tailoring γ_z and n_z and λ_z values as described above (doubling effective horizons until the maximum horizon is reached), still yields improvements without additional tuning.

As we mention in section 5.1, another benefit of TD(Δ) is the ability to examine the value function at different time-scales after a single pass through the network. That is, we can compose value functions from $\gamma_0, ..., \gamma_Z$ and understand the differences between different time-scales. This has implications for real-world uses with similar motivations as Sherstan et al. (2018) described. Take for example an MDP where the bulk of rewards are in some central region, requiring a policy π for some number of time-steps before reaching the dense reward region. By examining each w_z component in figure 5.1, a practitioner could understand how

far into a trajectory π must be followed before the dense reward region is reached. This adds some layer of interpretability to the value function which is missing in the single estimator case.

Throughout this work, we emphasize this algorithm as a complement to the selection of a final γ_Z. The longest horizon discount factor can be chosen according to other methods (hyperparameter optimization or meta-gradient methods). However, an added benefit of our method that is not explored in this work is its functionality as an *almost anytime* algorithm. While longer time horizons will take longer to converge, at any point in time, the sum of all horizons which have converged are a suitable approximation for the value function at that intermediary point. Therefore, with enough resources, TD(Δ) could potentially, at anytime, add one further time-scale $Z \leftarrow Z + 1$ (initialized to $w_{Z+1} = 0$ which preserves the current v estimate).

We have described the uses, theoretical properties, and empirical performance of TD(Δ). While we focus on PPO, our method could be dropped into many other algorithmic settings as a black-box, making it easy to use with clear performance benefits. We believe that TD(Δ) is an important drop-in addition to any TD-based training methods that can be applied to a number of existing model-free RL algorithms. We especially highlight the value of this method for performance tuning. We show that a simple sequence of γ_z values based on doubling horizon values can yield performance gains especially in dense settings, but this performance can be enhanced further with tuning. As the complexity of modeling and training long-horizon problems increases, TD(Δ) may be another tool for scaling and honing production systems for optimal performance.

Chapter 6

Bellman Decomposition Theorem

The works that have been presented up to this point have decomposed the Bellman equation in several ways. In this chapter, we present a unified equivalence theorem, that provides an equivalence between Bellman decompositions and standard TD learning. The key insight of the theorem is that if the sum of the decomposed TD errors equals the standard TD error, then there is a strict equivalence between TD and their decomposed counterparts. The equivalence is strict in the sense that they will produce identical parameters at every step of the learning process under certain assumptions. Interestingly, the equivalence extends to linear function approximation. Furthermore, the equivalence theorem is applicable to decompositions presented in chapter 3 (HRA) and chapter 5 (TD(Δ)).

6.1 Setup and Assumptions

We first restate the update function for temporal difference methods. Specifically, temporal-difference methods use a semi-gradient update function with the following structure:

$$\theta_{t+1} = \theta_t + \alpha \delta^{\lambda}_{t:t+n} \nabla_\theta q(s_t, a_t; \theta_t) \tag{6.1}$$

where $\delta^{\lambda}_{t:t+n}$ is the n-step truncated λ weighted TD error defined in equation 2.44. We note that any value function, v, can be cast as a q-function by assuming that there is only 1 action to be taken in the environment. Therefore, due to the increased generality, we will use q-functions in this section for the proof.

We consider decomposed methods that when summed together result in the original value

function. Formally, we use the following decomposition:

$$q(s,a) = \sum_{z=0}^{Z} w_z(s,a). \tag{6.2}$$

Furthermore, updates to the decomposed value function are performed as follows:

$$\theta^z_{t+1} = \theta^z_t + \alpha_z \delta^{\lambda,z}_{t:t+n} \nabla_\theta w(s_t, a_t; \theta^z_t), \tag{6.3}$$

where we purposely do not define δ^z, since this will be unique for each decomposition. We do note, however, that this encompasses methods from chapter 3 and chapter 5.

For example, we note that when training the decomposed value functions in HRA towards the optimal target from section 3.2.3 we have the following TD error using SARSA:

$$\delta^z_{t:t+1} = r^z_t + \gamma \hat{q}_z(s_{t+1}, a_{t+1}) - \hat{q}_z(s_t, a_t), \tag{6.4}$$

where we recall that $\sum_z r^z_t = r_t$ is the decomposed reward function.

Similarly, the decomposition found in TD(Δ) will also be analyzed in this chapter. In TD(Δ), we simply considered the value estimation problem and therefore the TD error was defined in terms of value functions. For consistency with this chapter, we define the TD error using SARSA:

$$\delta^z_{t+1} = (\gamma_z - \gamma_{z-1}) \sum_{u=0}^{z-1} \hat{w}_u(s_{t+1}, a_{t+1}) + \gamma_z \hat{w}_z(s_{t+1}, a_{t+1}) - \hat{w}_z(s_t, a_t). \tag{6.5}$$

Our main assumption is that the value function is modelled with linear function approximation.

Assumption 6.1.1.

$$q(s, \cdot) = \langle \theta, x(s) \rangle . \tag{6.6}$$

Similarly, the decomposed value functions also use a linear parameterization.

Assumption 6.1.2.

$$w_z(s, \cdot) = \langle \theta^z, x(s) \rangle . \tag{6.7}$$

The next assumption that is needed is that all of the decomposed value functions use the same learning rate as the non-decomposed variant.

Assumption 6.1.3.

$$\alpha_z = \alpha, \quad \forall z. \tag{6.8}$$

Our final assumption will be crucial to the induction step of the proof. It also will be the key to identifying whether a decomposition falls under the equivalence proof or not. Specifically, assuming that the sum of the decomposed value function is equal to some arbitrary reference value function, the sum of the decomposed TD errors has to be equal to the TD error of said reference value function.

Assumption 6.1.4. *Let* $q(s,a;\theta_t) = \sum_{z=0}^{Z} w_z(s,a;\theta_t^z)$ *then:*

$$\delta_{t:t+1} = \sum_{z=0}^{Z} \delta_{t:t+1}^z \quad \forall t, \tag{6.9}$$

where $\delta_{t:t+1}$ *is the one-step TD error defined by* $q(s,a;\theta_t)$ *and* $\delta_{t:t+1}^z$ *is the one-step TD error for* $w_z(s,a;\theta_t^z)$*.*

Crucially, the above assumption can be easily verified for reward decompositions such as HRA, assuming that the agents update towards the optimal target (Russell and Zimdars, 2003; Laroche et al., 2017). Moreover, we already showed that this assumption can be satisfied for TD(Δ) in theorem 5.3.1. For completeness we provide the following lemma for HRA.

Lemma 6.1.1. *Given updates according to equations 6.3 and 6.4, we have that assumption 6.1.4 holds, i.e., that:*

$$\delta_{t:t+1} = \sum_{z=0}^{Z} \delta_{t:t+1}^z \quad \forall t \tag{6.10}$$

Proof.

$$\begin{aligned}
\sum_{z=0}^{Z} \delta_t^z &= \sum_{z=0}^{Z} r_t^z + \gamma w_z(s_{t+1}, a_{t+1}; \theta_t^z) - \hat{w}_z(s_t, a_t; \theta_t^z) \\
&= r_t + \sum_{z=0}^{Z} \gamma w_z(s_{t+1}, a_{t+1}; \theta_t^z) - \hat{w}_z(s_t, a_t; \theta_t^z) && \text{(by decomposition)} \\
&= r_t + \gamma q(s_{t+1}, a_{t+1}; \theta_t) - q(s_t, a_t; \theta_t) = \delta_t && \text{(by assumption)}
\end{aligned} \tag{6.11}$$

□

Thus, if the parameters of the decomposed value functions using HRA produce the same value function as the non-decomposed counterpart, than the summed TD error for HRA is identical to the standard TD error.

6.2 Bellman Decomposition Theorem

With all the assumptions outlined in section 6.1, we state and prove the Bellman decomposition theorem.

Theorem 6.2.1. *If assumptions 6.1.1, 6.1.2, 6.1.3, and 6.1.4 hold, and if we pick the initial conditions such that* $\sum_{z=0}^{Z} \theta_0^z = \theta_0$*, then the iterates produced by equation 6.1 and equation 6.3 satisfy:*

$$\sum_{z=0}^{Z} \theta_t^z = \theta_t, \forall t, \tag{6.12}$$

Proof. The proof is by induction. We first need to show that the base case holds at $t = 0$. This is true given the assumption on initialization. We also note that the base case holds by using zero-initialization

Next, we assume that the statement holds for a given time-step t and show that it holds at next time-step $t + 1$.

$$\begin{aligned}
\sum_{z=0}^{Z} \theta_{t+1}^z &= \sum_{z=0}^{Z} \left(\theta_t^z + \alpha_z \delta_{t:t+n}^{\lambda,z} \nabla_{\theta^z} w_z(s_t, a_t; \theta^z) \right) \\
&= \sum_{z=0}^{Z} \left(\theta_t^z + \alpha_z \delta_{t:t+n}^{\lambda,z} x(s_t) \right) && \text{(by assumption 6.1.2)} \\
&= \theta_t + \left(\sum_{z=0}^{Z} \alpha_z \delta_{t:t+n}^{\lambda,z} \right) x(s_t) && \text{(by induction)} \\
&= \theta_t + \alpha \left(\sum_{z=0}^{Z} \delta_{t:t+n}^{\lambda,z} \right) x(s_t) && \text{(by assumption 6.1.3)} \\
&= \theta_t + \alpha \delta_{t:t+n}^{\lambda} x(s_t) && \text{(by assumption 6.1.4)} \\
&= \theta_{t+1}. && \text{(by assumption 6.1.1)}
\end{aligned} \tag{6.13}$$

□

Notably, using the same assumptions, the equivalence theorem is immediately applicable to both HRA and TD(Δ) using SARSA. This can be shown using lemma 6.1.1 for HRA and theorem 5.3.1 for TD(Δ). Moreover, the theorem immediately extends to the pure policy evaluation case as well as expected SARSA.

6.3 Discussion

The equivalence theorem provides insight into the conditions needed to have an equivalence between standard TD methods and methods that decompose the Bellman equation as described. While we have discussed two potential decompositions in this thesis, it is important to note that there are many other potential applications of this theorem. For example, in terms of the reward function, instead of the reward being decomposed based off of collectable objects, it can instead be decomposed based off of the sign, i.e., one value function for positive rewards and one for negative rewards. Another possibility is to use the magnitude, i.e., a value function for each differing reward magnitude. Moreover, instead of using the discount factor, the value function can be discretely decomposed based on distance in time, i.e., one value function that represents the sum of rewards over the next k time-steps, another for the next k steps, and so on. In any case, the importance of the equivalence theorem is not simply to identify the equivalence, but also to understand how to break the equivalence to potentially improve the sample efficiency over standard TD methods.

Since the equivalence only holds with linear function approximation, the most obvious way to break the equivalence is through the use of neural networks as a function approximator. The network structure that we used in chapter 5, with a shared lower representation and a linear mapping to the decomposed value functions, is only one possible architecture, and others could be considered. For example, one possibility, is to use smaller networks to estimate the value functions of simpler decompositions (if this is known to the practitioner beforehand). In the case of the decomposition based on the discount factor in chapter 5, value functions based on a smaller γ could have been trained with smaller networks due to their simplicity, and vice versa. However, for other decompositions this may not be immediately obvious to the practitioner beforehand. For example, in the reward decomposition presented in chapter 3, it is unclear which of the rewards in Ms. Pacman could benefit from a larger or smaller network.

As previously discussed in chapter 5, another way to break the equivalence is to use different n-step or λ-returns for each decomposed value function. While we were able to theoretically analyze reducing the n-step hyperparameter for value functions with a smaller value of γ in thereom 5.3.4, this may not be obvious for other decompositions. Finally, we can also break the equivalence by using different learning rates for each decomposed value function. Unfortunately, it may also be difficult to know in advance what learning rate to set for each value function. However, as previously discussed in chapter 5, adaptive optimization techniques use a per parameter learning rate and will automatically break the equivalence. We further explore adaptive optimization techniques for TD learning in the following chapter.

Chapter 7

Jacobi Preconditioning for TD

One of the core elements of the Bellman decomposition theorem (theorem 6.2.1), is the assumption that the same learning rate is used for every decomposed value function. As alluded to, one way to break the equivalence is to use an adaptive per-parameter learning rate, as existing adaptive optimizers do (Tieleman and Hinton, 2012; Kingma and Ba, 2014). Most adaptive optimizers, however, are built with supervised learning in mind and do not explicitly account for the TD case. Previous work has investigated whether adaptive optimizers can be constructed that are better suited for TD learning (Henderson et al., 2018; Sun et al., 2020).

We propose to use the Jacobi preconditioner (Greenbaum, 1997), a diagonal approximation of the optimal gain matrix, which results in an efficient and principled adaptive method for TD, which we call TDprop. We theoretically compare the approach in the tabular setting against standard TD methods. We also show how this method can be easily adapted to the deep RL setting and compare and contrast it with other deep learning optimizers. Surprisingly, we find that both theoretically and empirically, after a hyperparameter search, TDprop behaves similarly to other optimizers, both TDProp and SGD meet or exceed the performance of Adam (Kingma and Ba, 2014). This result suggests that while Jacobi preconditioning may be an improved approach to adaptive optimization in deep TD learning, further work is needed for adaptive optimization methods to yield a strict improvement over SGD.

7.1 Related Work

Devraj and Meyn (2017) derived and studied using the optimal gain matrix from stochastic approximation (Benveniste et al., 2012) for linear TD learning. They found that in the linear case, the optimal gain matrix directly corresponds with the *least squares temporal-difference* (LSTD) method (Boyan, 1999). Recently, the approach was extended to the non-linear function approximation setting (Chen et al., 2019). Unlike those methods, we propose to use

a diagonal approximation of the gain matrix. This change provides us with a computationally tractable approach that can scale to millions of parameters, as is common in the deep RL setting.

A wide range of work has examined adaptive optimization and preconditioners in supervised learning. For example, LeCun et al. (2012) describe the benefits of the Jacobi preconditioner as well as efficient implementations. Schaul et al. (2013) propose a method for adaptively tuning both the global learning rate, as well as the per parameter learning rates based off both the Jacobi preconditoner and local variance of the gradient. Dauphin et al. (2015) discuss trade-offs between the Jacobi preconditioner and the *equilibriated* preconditioner (which has similar properties to popular methods such as RMSprop (Tieleman and Hinton, 2012) and Adam (Kingma and Ba, 2014)). Finally, Martens (2014) presents a unified view of methods such as RMSprop and Adam for approximating the empirical Fisher matrix. Recently, Sun et al. (2020) extended the adaptive update rule from Duchi et al. (2011) to the TD setting and studied its convergence properties. Their theoretical results validate the use of standard adaptive optimizers from the deep learning literature in the TD setting. We compare and contrast our method to state of the art deep learning optimizers in section 7.8.

There has been a vast array of work that explored adaptive optimizers and preconditioners for linear TD learning. For example, *scalar incremental delta-bar-delta* (SID) (Dabney, 2014) extend *incremental delta-bar-delta* (IDBD) (Sutton, 1992) to linear TD and adaptively tune a single global learning rate. Similarly, (Dabney and Barto, 2012) derive and examine an optimal global (not per parameter) learning rate for linear TD. Recently, *TD incremental delta-bar-delta* (TIDBD) (Kearney et al., 2019), adaptively learn a per parameter learning rate based on the correlation between state features and TD errors. To our knowledge, however, TIDBD has not been extended to the non-linear setting with TD learning. In terms of preconditioners, Yao and Liu (2008) present a generalized framework for using varying preconditioners in TD learning and subsequently Yao et al. (2009) propose to use the full matrix H^{-1} as a preconditioner for linear TD learning. Perhaps the closest works to our own are approaches based on approximating H, as in Givchi and Palhang (2015); Pan et al. (2017). Both works examine the use of the diagonal approximation (among other approximations), however, in both cases the design of the algorithm, and the empirical analysis is restricted to the linear setting. We expand on the theoretical linear analysis proposed in these works and provide empirical evidence in deep RL settings.

Finally, in the tabular case, Jacobi preconditioning can be interpreted as a per state learning rate based on a partial model of the world dynamics. Similarly, performing expected TD updates using a learned model of the transition dynamics has been shown to improve sample efficiency in both the tabular (Sutton, 1991) and the deep RL setting (Feinberg et al.,

2018). Unlike those methods, Jacobi preconditioning does not plan with the learned model and only requires the tracking of a partial model of the dynamics, i.e., the probability of remaining in the same state.

7.2 Optimal Gain Matrix

We recall that in stochastic approximation, updates are iteratively performed to a parameter vector:

$$\theta_{t+1} = \theta_t + \alpha_{t+1} g(\theta_t), \tag{7.1}$$

where α is the learning rate and $g(\theta_t)$ is the update function.

The optimal gain matrix (learning rate), in terms of asymptotic convergence properties, is the negative of the inverse gradient of the expected update function (Benveniste et al., 2012):

$$H^{-1} = -(\nabla_\theta \mathbb{E}^\mu [g(\theta_t)])^{-1}, \tag{7.2}$$

where μ is the stationary distribution and $H^{-1} \in \mathbb{R}^{d \times d}$ is the resulting matrix gain.

The update to the parameters then becomes:

$$\theta_{t+1} = \theta_t + \alpha_{t+1} H^{-1} g(\theta_t), \tag{7.3}$$

where H^{-1} is considered to be a preconditioner (Greenbaum, 1997). Moreover, the choice of notation for H is intentional, as in gradient descent H corresponds with the *Hessian* of the loss function. In the following sections we explore the use of approximations of the optimal gain matrix for both regression and TD learning.

7.3 Jacobi Preconditioning for Regression

In the following sections, we compare and contrast Jacobi preconditioning for TD learning and supervised regression. To this end, we first present the common *sum of squares* error function:

$$\mathcal{L}(\theta) = \mathbb{E}^\mu \left[\frac{1}{2} (\hat{y}_t - y_t)^2 \right], \tag{7.4}$$

where $\hat{y}_t = f(x_t; \theta_t)$ is the estimate given the input x_t and parameters θ_t, and y_t is the target at time t.

The expected update direction is the negative gradient of $\mathcal{L}(\theta)$:

$$\mathbb{E}^\mu [g(\theta)] = -\nabla_\theta \mathcal{L}(\theta) = -\mathbb{E}^\mu [\delta_t \nabla_\theta \hat{y}_t], \tag{7.5}$$

where $\delta_t = \hat{y}_t - y_t$ is the error at time t.

The corresponding negative gradient of the update direction (i.e., the Hessian of the loss function) is then:

$$\begin{aligned} H &= -\nabla_\theta(-\nabla_\theta \mathcal{L}(\theta)) = \nabla^2_\theta \mathcal{L}(\theta) \\ &= \mathbb{E}^\mu \left[\nabla_\theta \hat{y}_t \nabla_\theta \hat{y}_t^\top + \delta_t \nabla^2_\theta \hat{y}_t \right], \end{aligned} \tag{7.6}$$

where ∇^2 corresponds to applying the gradient operator twice. Estimating the full Hessian can be computationally intractable due to the second order terms from equation (7.6). Instead, the outer product approximation, also known as the *Gauss Newton* approximation (Bishop, 2006), drops the second order terms from equation (7.6):

$$H = \nabla^2_\theta \mathcal{L}(\theta) \approx \mathbb{E}^\mu \left[\nabla_\theta \hat{y}_t \nabla_\theta \hat{y}_t^\top \right]. \tag{7.7}$$

Finally, to obtain a per-parameter learning rate, we can approximate the Hessian matrix by its diagonal:

$$\bar{H} \approx \mathbb{E}^\mu \left[\text{diag}(\nabla_\theta \hat{y}_t \nabla_\theta \hat{y}_t^\top) \right], \tag{7.8}$$

which is known as the *Jacobi preconditioner* (Greenbaum, 1997). Specifically, updates to the parameter vector are performed via the following:

$$\theta_{t+1} = \theta_t + \alpha_{t+1} \bar{H}^{-1} g(\theta_t). \tag{7.9}$$

The main benefit of the diagonal approximation is that estimating and inverting the gain matrix is significantly cheaper computationally. The approximation accuracy will depend greatly on the problem at hand; nevertheless, both its low space and computational complexity has led to its usage (LeCun et al., 2012).

7.4 Jacobi Preconditioning for TD

We first recall that given the semi-gradient update function defined in equation (2.43), we have the following:

$$g(\theta_t) = \delta^\lambda_{t:t+n} \nabla_\theta \hat{v}^\pi(s_t; \theta_t), \tag{7.10}$$

where $\delta^\lambda_{t:t+n} = G^\lambda_{t:t+n} - \hat{v}^\pi(s_t; \theta_t)$ is the TD error at time t.

For a direct comparison to supervised regression, we set $\hat{y}_t = \hat{v}^\pi(s_t; \theta_t)$ and arrive at the

following calculation for the negative gradient of the update function, H:

$$\begin{aligned} H &= -\nabla_\theta \mathbb{E}^\mu \left[\delta^\lambda_{t:t+n} \nabla_\theta \hat{y}_t \right] \\ &= -\mathbb{E}^\mu \left[\nabla_\theta \delta^\lambda_{t:t+n} \nabla_\theta \hat{y}_t^\top + \delta^\lambda_{t:t+n} \nabla^2_\theta \hat{y}_t \right]. \end{aligned} \tag{7.11}$$

To obtain an efficient adaptive optimizer we propose to use the diagonal approximation (the Jacobi preconditioner) as described in equation (7.8):

$$\bar{H} \approx -\mathbb{E}^\mu \left[\text{diag}(\nabla_\theta \delta^\lambda_{t:t+n} \nabla_\theta \hat{y}_t^\top) \right]. \tag{7.12}$$

To compare this expression to what was obtained for supervised regression in equation (7.8), we can expand the outer product:

$$\bar{H} = \mathbb{E}^\mu \Bigg[\text{diag} \left(\nabla_\theta \hat{y}_t \nabla_\theta \hat{y}_t^\top - \lambda^{n-1} \gamma^n \nabla_\theta \hat{y}_{t+n} \nabla_\theta \hat{y}_t^\top + \sum_{k=1}^{n-1} (\gamma\lambda)^{k-1} (\gamma\lambda - \gamma) \nabla_\theta \hat{y}_{t+k} \nabla_\theta \hat{y}_t^\top \right) \Bigg], \tag{7.13}$$

where we note that the left most term $\nabla_\theta \hat{y}_t \nabla_\theta \hat{y}_t^\top$ is the same as the diagonal outer product approximation that arises from the sum of squares loss function in equation (7.8). The remaining terms are unique to temporal-difference learning. Moreover, the terms inside the summation disappear when $\lambda = 1$ (i.e., when not using λ-returns).

7.5 Interesting Cases

The following section discusses some interesting sub-cases of the Jacobi preconditioner.

TD(0): For the special case where $\lambda = 0$ we have:

$$\nabla_\theta \delta^{\lambda=0}_{t:t+1} = \gamma \nabla_\theta \hat{y}_{t+1} - \nabla_\theta \hat{y}_t, \tag{7.14}$$

plugging this back into equation (7.11) and using the diagonal outer product approximation:

$$\bar{H} = \mathbb{E}^\mu \left[\text{diag} \left(\nabla_\theta \hat{y}_t \nabla_\theta \hat{y}_t^\top - \gamma \nabla_\theta \hat{y}_{t+1} \nabla_\theta \hat{y}_t^\top \right) \right], \tag{7.15}$$

which resembles the standard outer product approximation of the sum of squares loss functions in equation (7.7) with an additional correction term that depends on the product of gradients of successive value functions.

Tabular TD(0) Case: In the tabular case with TD(0) we have:

$$\begin{aligned}\bar{H}_{i,i} &= \left[\mathbb{E}^{\mu}\left[\operatorname{diag}\left(\nabla_{\theta}\hat{y}_t\nabla_{\theta}\hat{y}_t^{\top} - \gamma\nabla_{\theta}\hat{y}_{t+1}\nabla_{\theta}\hat{y}_t^{\top}\right)\right]\right]_{i,i} \\ &= \mu(s^i)\left[\nabla_{\theta^i}\hat{v}(s^i;\theta^i)\nabla_{\theta^i}\hat{v}(s^i;\theta^i) - \sum_{s'} p(s'|s^i,\pi)\gamma\nabla_{\theta^i}\hat{v}(s';\theta^i)\nabla_{\theta^i}\hat{v}(s^i;\theta^i)\right], \end{aligned} \tag{7.16}$$

which can be simplified by noticing that in the tabular case $\nabla_{\theta^i}\hat{v}(s^i;\theta^i) = 1$ and that $\nabla_{\theta^i}\hat{v}(s';\theta^i)$ is 0 when $s' \neq s$ and 1 when $s = s'$. In terms of $\bar{H}^{-1}$ this gives us:

$$\bar{H}^{-1}_{i,i} = \frac{1}{\mu(s^i)(1-\gamma p(s'=s|s=s^i,\pi))}. \tag{7.17}$$

This can be interpreted as a per state learning rate that is reweighted by both the stationary distribution and the probability of self-looping, i.e., the probability of remaining in the current state.

TD(1) / Target Network: Another interesting case is when $\lambda = 1$ and $n = \infty$ (i.e., TD(1) or Monte-Carlo) or when using a target network, we get the following outer product approximation:

$$\bar{H} = \mathbb{E}^{\mu}\left[\operatorname{diag}\left(\nabla_{\theta}\hat{y}_t\nabla_{\theta}\hat{y}_t^{\top}\right)\right], \tag{7.18}$$

which is the same H as the sum of squares loss function. We can interpret this similarity as suggesting that as $n \to \infty$, or when using a target network, TD learning approaches supervised learning. Intuitively, when doing policy evaluation with a fixed policy, regressing towards the Monte-Carlo return is very similar to standard supervised learning. Moreover, when using a target network, if we presume that the network is updated slowly enough, then the targets appear to be fixed for the agent, making it also very similar to supervised learning.

7.6 Theoretical Analysis

In this section, we prove certain convergence properties of applying the Jacobi preconditioner to TD learning, following a similar analysis to that of Schoknecht and Merke (2003). Specifically, we aim to solve the following linear equation:

$$r + (\gamma P - I)v = 0, \tag{7.19}$$

where $r \in \mathbb{R}^{|S|}$ is the expected reward vector, $P \in \mathbb{R}^{|S|\times|S|}$ is the transition matrix and $v \in \mathbb{R}^{|S|}$ is the estimated value function. We note that $r + (\gamma P - I)v = 0$ at the solution, i.e.,

when $v = v^*$.

Moreover, the iterative update procedure, using a constant learning rate, can be described by the following equation:

$$v_{t+1} = v_t - \alpha \left(H v_t - r \right), \tag{7.20}$$

where $H = \text{diag}(I - \gamma P)$.

Next, we determine the asymptotic convergence rate of this iterative update, similar to the analysis provided in section 2.1, except with a constant learning rate. Specifically, by defining the error vector as $e_t = v_t - v^*$, where for v^* we have that $Hv^* - r = 0$, we can derive the following recursion:

$$\begin{aligned} e_{t+1} &= v_{t+1} - v^* \\ &= (I - \alpha H) v_t + \alpha r - v^* + \underbrace{\alpha (H v^* - r)}_{=0} \\ &= (I - \alpha H)(v_t - v^*) = (I - \alpha H)^{t+1} e_0, \end{aligned} \tag{7.21}$$

thus the asymptotic convergence rate is $\rho(I - \alpha H)$.

Applying the Jacobi preconditioner to the original system we get the following iterative formula:

$$v_{t+1} = v_t - \alpha \bar{H}^{-1} \left(H v_t - r \right) \tag{7.22}$$

where following equation 7.12, we have $\bar{H} = \text{diag}(H) = \text{diag}(I - \gamma P)$. We also note that the asymptotic convergence rate of the preconditioned system is $\rho(I - \alpha \bar{H}^{-1} H)$. In the following subsection, we introduce the theory of regular splittings to further analyze the Jacobi preconditioner.

7.6.1 Comparing Regular Splittings

Using the theory of regular splittings (Varga, 1962) we can frame both the Jacobi preconditioner and the original system as regular splittings and thereby prove that it has a better convergence rate.

Definition 7.6.1. *(regular splitting: definition 3.28 from Varga (1962)) If $H = B - C$, $B^{-1} \geq 0$, and $C \geq 0$ for all components, then $B - C$ is said to be a regular splitting of H.*

Moreover, we have the following proposition that allows us to compare the asymptotic convergence rates of different regular splittings.

Proposition 7.6.1. *(comparing regular splittings: theorem 3.32 from Varga (1962) and corollary 10.3.1 from Greenbaum (1997)): Let (B_1, C_1), and (B_2, C_2) be regular splittings of*

H. Then if $H^{-1} \geq 0$ and $0 \leq C_2 \leq C_1$ for all components, then:

$$0 \leq \rho(B_2^{-1}C_2) \leq \rho(B_1^{-1}C_1) < 1. \tag{7.23}$$

Moreover, if $H^{-1} > 0$ and $0 \leq C_2 \leq C_1$ for all components and $C_1 \neq 0$, $C_2 \neq 0$, and $C_2 - C_1 \neq 0$, then

$$0 < \rho(B_2^{-1}C_2) < \rho(B_1^{-1}C_1) < 1. \tag{7.24}$$

Following definition 7.6.1, and using $H = I - \gamma P$, the Jacobi preconditioner can be seen as a regular splitting $H = \bar{B} - \bar{C}$ where $\bar{B} = \bar{H}$ and $\bar{C} = \bar{B} - H$. Similarly, for standard TD we have that $H = B - C$ where $B = I$ and $C = \gamma P$ forms a valid regular splitting of H. With both methods framed in terms of regular splittings, we can now compare their convergence rates by using proposition 7.6.1 and setting the learning rate to 1, i.e., the standard iterative setting from section 2.1.

Theorem 7.6.1. *Let $H = I - \gamma P$ and $\bar{H} = \text{diag}(H)$, then we have that:*

$$\rho(I - \bar{H}^{-1}H) \leq \rho(I - H) < 1. \tag{7.25}$$

Proof. We can rewrite $\rho(I - \bar{H}^{-1}H) = \rho(\bar{H}^{-1}\bar{C})$ and $\rho(I - H) = \rho(C)$. The rest of the proof follows directly from the properties of regular splittings from proposition 7.6.1. □

The previous theorem omitted the use of learning rates, in fact, it explicitly assumed a learning rate of 1. In the following section, we reintroduce the learning rate to our analysis, to account for the fact that in practice the learning rate is typically tuned to provide optimal results.

7.6.2 Reintroducing the Learning Rate

Our analysis for tuning the learning rate will be limited to symmetric transition matrices, due to the fact that they are guaranteed to have real eigenvalues (Greenbaum, 1997). While this is indeed a limiting assumption that does not apply to every MDP, we still find that the results are interesting. Specifically, we find that once the learning rate has been tuned, that we cannot guarantee that for every symmetric MDP that Jacobi preconditioning provides an improvement in terms of the convergence rate.

To determine which method has the best convergence rate after the learning rate has been tuned, we define the optimal spectral radius as the minimum over all feasible α.

Definition 7.6.2. *Given a matrix H with only positive real eigenvalues $eig(H) \in \mathbb{R}^{>0}$ we define the optimal learning rate as:*

$$\alpha^* := \arg\min_{\alpha} \rho(I - \alpha H) \tag{7.26}$$

and the resulting optimal spectral radius $\rho^*(I - \alpha H)$:

$$\rho(I - \alpha^* H) := \min_{\alpha} \rho(I - \alpha H) \tag{7.27}$$

We now focus on finding α^* and the resulting ρ given H. First, by standard eigenvalue properties we have the following fact.

Fact 7.6.1. *The eigenvalues of $(I - \alpha H)$ can be rewritten as:*

$$eig(I - \alpha H) = 1 - \alpha eig(H). \tag{7.28}$$

Next, we rewrite $\rho(I - \alpha H)$ in terms of just the eigenvalues of H.

Lemma 7.6.1. *For a fixed $\alpha > 0$ and a matrix H we have that:*

$$\rho(I - \alpha H) = \max_{\lambda^H} \left|1 - \alpha \lambda^H\right|. \tag{7.29}$$

Proof. The proof comes directly from fact 7.6.1 and the definition of the spectral radius in definition 2.1.1. □

Next, we rewrite $\rho(I - \alpha H)$ in terms of just the maximum and minimum eigenvalues of H.

Lemma 7.6.2. *For a fixed $\alpha > 0$ and a matrix H with only positive real eigenvalues $eig(H) = \{\lambda_1, \lambda_2, ...\} \in \mathbb{R}^{>0}$ we have that:*

$$\rho(I - \alpha H) = \max\left\{\left|1 - \alpha \lambda^H_{max}\right|, \left|1 - \alpha \lambda^H_{min}\right|\right\}. \tag{7.30}$$

Proof. The proof follows directly from proposition 7.6.1 and the assumption that all of the eigenvalues of H are positive. □

In words, we have that the spectral radius of $I - \alpha H$ is either a function of the minimum or the maximum eigenvalue of H. Finally, the optimal learning rate α^*, for the simple case where $eig(H) \in \mathbb{R}$, is derived in the following proposition.

Proposition 7.6.2. *(corollary 1 from Schoknecht and Merke (2003)) For a matrix H with only positive real eigenvalues $eig(H) = \{\lambda_1, \lambda_2, ...\} \in \mathbb{R}^{>0}$ we have that the α that corresponds to $\min_\alpha \rho(I - \alpha H)$ is:*

$$\alpha^* = \frac{2}{\lambda^H_{max} + \lambda^H_{min}} \tag{7.31}$$

and the resulting spectral radius:

$$\rho(I - \alpha^* H) = \frac{\lambda^H_{max} - \lambda^H_{min}}{\lambda^H_{max} + \lambda^H_{min}} = \frac{\kappa(H) - 1}{\kappa(H) + 1}. \tag{7.32}$$

where $\kappa = \frac{\lambda^H_{max}}{\lambda^H_{min}}$ is the condition number.

Proof. We recall that from lemma 7.6.2, we have that:

$$\rho(I - \alpha H) = \max\left\{\left|1 - \alpha\lambda^H_{max}\right|, \left|1 - \alpha\lambda^H_{min}\right|\right\}.$$

The α that minimizes this will have the minimal and maximal eigenvalues symmetrically around zero. Therefore $-(1 + \alpha^*\lambda^H_{max}) = (1 + \alpha^*\lambda^H_{min})$ which implies that:

$$\alpha^* = \frac{2}{\lambda^H_{max} + \lambda^H_{min}}.$$

Moreover, plugging α^* into the $\rho(I - \alpha^* H)$ gives us the desired spectral radius. □

We highlight that from proposition 7.6.2, the *optimal spectral radius* is a monotonically increasing function of the *condition number*, which means that poorly conditioned matrices will induce a slow convergence. As a result, we seek to reduce the condition number of H with the Jacobi preconditioner. By comparing the condition numbers of the Jacobi preconditioned TD and standard TD, we can determine which method has better convergence properties and performs best under their respective optimal learning rates.

To do so, we first have the following theorem that tells us about the spectral radius of positive matrices.

Theorem 7.6.2. *(Theorem 10.2.4 from Greenbaum (1997)) If $A \in \mathbb{R}^{n\times n}$ and $A \geq 0$ for all components, then $\rho(A)$ is an eigenvalue of A and there is a nonnegative vector $v \geq 0$ with $\|v\| = 1$, such that $Av = \rho(A)v$.*

Thus, for positive matrices we have that the spectral radius corresponds to a simple eigenvalue. We now compare the condition number between two regular splittings with the following two theorems from Greenbaum (1997), which should be read consecutively.

Theorem 7.6.3. *(theorem 10.4.1 from Greenbaum (1997)) Let H, B_1, and B_2 be symmetric positive-definite matrices satisfying the assumptions of proposition 7.6.1 and suppose that the largest eigenvalue of $B_2^{-1}H \geq 1$. Then the ratios of the largest to smallest eigenvalues of $B_1^{-1}H$ and $B_2^{-1}H$ satisfy:*

$$\frac{\lambda_{max}^{B_1^{-1}H}}{\lambda_{min}^{B_1^{-1}H}} \leq 2\frac{\lambda_{max}^{B_2^{-1}H}}{\lambda_{min}^{B_2^{-1}H}}. \tag{7.33}$$

Proof. Since the elements of $B_2^{-1}C_2$ are nonnegative (from definition 7.6.1), it follows from theorem 7.6.2 that its spectral radius corresponds to its largest eigenvalue (which is also simple):

$$\rho(B_2^{-1}C_2) = \rho(I - B_2^{-1}H) = 1 - \lambda_{min}^{B_2^{-1}H}.$$

The result $\rho(B_1^{-1}C_1) \leq \rho(B_2^{-1}C_2)$ from proposition 7.6.1 implies that:

$$1 - \lambda_{min}^{B_1^{-1}H} \leq 1 - \lambda_{min}^{B_2^{-1}H} \text{ and } \lambda_{max}^{B_1^{-1}H} - 1 \leq 1 - \lambda_{min}^{B_2^{-1}H}$$

or equivalently,

$$\lambda_{min}^{B_1^{-1}H} \geq \lambda_{min}^{B_2^{-1}H} \text{ and } \lambda_{max}^{B_1^{-1}H} \leq 2 - \lambda_{min}^{B_2^{-1}H}$$

Dividing the second inequality by the first gives:

$$\frac{\lambda_{max}^{B_1^{-1}H}}{\lambda_{min}^{B_1^{-1}H}} \leq \frac{\lambda_{max}^{B_2^{-1}H}}{\lambda_{min}^{B_2^{-1}H}} \left(\frac{2 - \lambda_{min}^{B_2^{-1}H}}{\lambda_{max}^{B_2^{-1}H}} \right). \tag{7.34}$$

Since by assumption $\lambda_{max}^{B_1^{-1}H} \geq 1$ and since $\rho(B_2^{-1}C_2) < 1$ implies that $\lambda_{min}^{B_2^{-1}H} > 0$, the second factor on the right-hand side is less than 2. □

Theorem 7.6.4. *(theorem 10.4.2 from Greenbaum (1997)) Let H, B_1, and B_2 be symmetric positive-definite matrices satisfying the assumptions of proposition 7.6.1, then the assumption in theorem 7.6.3 that the largest eigenvalue of $B_2^{-1}H \geq 1$ is satisfied if H and B_2 have at least one diagonal element in common.*

Proof. If H and B_2 have a diagonal element in common , then the matrix C_2 has a zero diagonal element (since $H = B_2 - C_2$). This implies that $B_2^{-1}C_2$ has a nonpositive eigenvalue since the smallest eigenvalue of this matrix satisfies:

$$\lambda_{\min}^{B_2^{-1}C_1} = \inf_{v \neq 0} \frac{v^\top C_2 v}{v^\top B_2 v} \leq \frac{u^\top C_2 u}{u^\top B_2 u} = 0$$

where u is the vector with a 1 in the position of the zero diagonal element and zeros elsewhere. Therefore $B_2^{-1}H = I - B_2^{-1}C_2$ has an eigenvalue greater than or equal to 1 since:

$$\lambda_{\max}^{B_2^{-1}H} = \sup_{v \neq 0} \left\{ 1 - \frac{v^\top C_2 v}{v^\top B_2 v} \right\} = 1 - \inf_{v \neq 0} \left\{ \frac{v^\top C_2 v}{v^\top B_2 v} \right\} = 1 - \underbrace{\lambda_{\min}^{B_2^{-1}C_1}}_{\leq 0} \geq 1$$

□

Theorems 7.6.3 and 7.6.4 show that once a pair of regular splittings have been scaled such that C_2 has been multiplied by a constant (which does not affect the condition number) that makes C_2 have a diagonal element in common with H, then the splitting with better convergence rate (in terms of proposition 7.6.1) has a condition number at worst two times the other. We now apply these results to compare the Jacobi preconditioned system to the original.

Theorem 7.6.5. *Let $H = (I - \gamma P)$ and $\bar{H} = diag\,(I - \gamma P)$, then assuming that H is symmetric we have that:*

$$\kappa\left(\bar{H}^{-1}H\right) \leq 2\kappa\left(H\right). \tag{7.35}$$

Proof. The result follows directly from theorems 7.6.3 and 7.6.4, and setting $A = H$, $B_1 = \bar{H}$, $B_2 = I$. □

In words, when H is symmetric, which is notably not all MDPs, the condition number of the Jacobi preconditioned system is at most a constant factor of 2 worse than the original system. The symmetric assumption is indeed limiting and only applies to MDPs with symmetric transition matrices. In practice, even though the theory does not extend to every MDP, we hypothesize that once a hyperparameter search has been conducted over the learning rate, the Jacobi preconditioner will have similar performance to the original system. Next, we provide extensions to both n-step and λ-returns.

7.6.3 Extension to n-step and λ returns

In the case of n-step and λ-returns we can derive analogous results to the single step case. This can be done by framing both the n-step and λ Jacobi preconditioning as regular splittings of their respective linear systems.

For n-step returns we have the following iterative update:

$$v_{t+1} = v_t - \alpha \left(H_n v_t - \sum_{k=0}^{n-1} (\gamma P)^k r \right), \tag{7.36}$$

where α is the learning rate, $H_n = (I - \gamma^n P^n)$, and v_t is the estimated value function at time t. By defining the error vector as before, $e_t = v_t - v^*$, we have that $e_{t+1} = (I - \alpha H_n)^{t+1} e_0$.

By applying the Jacobi preconditioner to the n-step system we get the following iterative formula:

$$v_{t+1} = v_t - \alpha \bar{H}_n^{-1} \left(H_n v_t - \sum_{k=0}^{n-1} (\gamma P)^k r \right) \tag{7.37}$$

where following equation 7.12, we have $\bar{H}_n = \text{diag}(H_n) = \text{diag}\,(I - \gamma^n P^n)$. We also note that the asymptotic convergence rate of the preconditioned system is $\rho(I - \alpha \bar{H}_n^{-1} H_n)$.

Following definition 7.6.1, and using $H_n = (I - \gamma^n P^n)$, the Jacobi preconditioner can be seen as a regular splitting $H_n = \bar{B}_n - \bar{C}_n$ where $\bar{B}_n = \bar{H}_n$ and $\bar{C}_n = \bar{B}_n - H_n$. Similarly, for standard n-step TD we have that $H_n = B_n - C_n$ where $B_n = I$ and $C_n = I - H_n$ forms a valid regular splitting of H_n. Since the requirements for proposition 7.6.1 apply ($H_n^{-1} \geq 0$ and $0 \leq \bar{C}_n \leq C_n$), under the same symmetric assumption required for theorem 7.6.3, analogous results for theorems 7.6.1 and 7.6.5 for the n-step preconditioned system also hold.

For λ-returns we have the following iterative update:

$$v_{t+1} = v_t - \alpha \left(H_\lambda v_t - (I - \gamma\lambda P)^{-1} r \right), \tag{7.38}$$

where α is the learning rate, $H_\lambda = (I - \gamma\lambda P)^{-1} (I - \gamma P)$, and v_t is the estimated value function at time t. By defining the error vector as before, $e_t = v_t - v^*$, we have that $e_{t+1} = (I - \alpha H_\lambda)^{t+1} e_0$.

By applying the Jacobi preconditioner to the λ linear system we get the iterative formula:

$$v_{t+1} = v_t - \alpha \bar{H}_\lambda^{-1} \left(H_\lambda v_t - (I - \gamma\lambda P)^{-1} r \right), \tag{7.39}$$

where following equation 7.12, we have $\bar{H}_\lambda = \text{diag}(H_\lambda) = \text{diag}\left((I - \gamma\lambda P)^{-1} (I - \gamma P) \right)$. We also note that the asymptotic convergence rate of the preconditioned system is $\rho(I - \alpha \bar{H}_\lambda^{-1} H_\lambda)$.

Following definition 7.6.1, and using $H_\lambda = (I - \gamma\lambda P)^{-1} (I - \gamma P)$, the Jacobi precondi-

tioner can be seen as a regular splitting $H_\lambda = \bar{B}_\lambda - \bar{C}_\lambda$ where $\bar{B}_\lambda = \bar{H}_\lambda$ and $\bar{C}_\lambda = \bar{B}_\lambda - H_\lambda$. Similarly, for standard TD(λ) we have that $H_\lambda = B_\lambda - C_\lambda$ where $B_\lambda = I$ and $C_\lambda = I - H_\lambda$ forms a valid regular splitting of H_λ. Since the requirements for proposition 7.6.1 apply ($H_\lambda^{-1} \geq 0$ and $0 \leq \bar{C}_\lambda \leq C_\lambda$), under the same symmetric assumption required for theorem 7.6.3, analogous results for theorems 7.6.1 and 7.6.5 for the λ system also hold.

7.7 Practical Implementation

In this section, we provide details for our practical algorithm, called TDprop, that tracks a per parameter learning rate based on the diagonal outer product approximation. Specifically, for each parameter i we have:

$$\bar{H}^{i,i} = z^i = -\mathbb{E}^{\mu_\theta}\left[\nabla_{\theta_i}\delta^\lambda_{t:t+n}\nabla_{\theta_i}\hat{y}_t\right]. \tag{7.40}$$

In practice, we use $|\bar{H}|$ because in non-convex optimization H might be indefinite, see Dauphin et al. (2015). Moreover, we found in initial testing that tracking $\bar{H}$ and then computing $|\bar{H}|$, led to poor performance due to the cancellation of positive and negative samples. Instead, to track z we compute an exponential moving average of the squared sampled statistic:

$$z_{t+1} = \beta z_t + (1-\beta)(-\nabla_\theta \delta^\lambda_{t:t+n} \odot \nabla_\theta \hat{y}_t)^2, \tag{7.41}$$

where $\odot$ is the element-wise product and $\beta \in [0, 1)$ is the tracking hyperparameter.

We then update the parameter vector θ using the square root of z:

$$\theta_{t+1} = \theta_t + \alpha\left(Z_{t+1}^{\frac{1}{2}} + \epsilon I\right)^{-1}\delta^\lambda_{t:t+n}\nabla_\theta \hat{y}_t, \tag{7.42}$$

where Z_{t+1} is the diagonal matrix formed from the elements of the vector z_{t+1}, α is the global learning rate, and ϵ is a damping hyperparameter.

In the mini-batch setting, TDprop needs to compute the required statistic $(-\nabla_\theta \delta^\lambda_{t:t+n} \odot \nabla_\theta \hat{y}_t)$ for each sample. Naively, this would increase the computation time by the size of the mini-batch. To alleviate this cost, we parallelize the computation with backpack (Dangel et al., 2019), a package for pytorch (Paszke et al., 2019). We provide pseudo-code for TDprop in algorithm 5.

Algorithm 5 *TDprop*

Require: α: Learning rate

Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates

Require: $\epsilon \in (0, 1]$: Damping Hyperparameter

$\bar{g}_0 \leftarrow 0$

$z_0 \leftarrow 1$

function UPDATE(δ_t, v_t, θ_t) (TD error δ_t, value function v_t, and parameters θ_t)

$\bar{g}_{t+1} \leftarrow \beta_1 \cdot \bar{g}_t + (1 - \beta_1) \cdot \delta_t \nabla_\theta v_t$

$z_{t+1} \leftarrow \beta_2 \cdot z_t + (1 - \beta_2) \cdot (\nabla_\theta \delta_t \odot \nabla_\theta v_t)^2$

$\theta_{t+1} \leftarrow \theta_t - \alpha \cdot \bar{g}_{t+1} / (\sqrt{z_{t+1}} + \epsilon)$

end function

7.8 Experiments

We perform a random hyperparameter search for TDprop, Adam (Kingma and Ba, 2014), as well as vanilla stochastic gradient descent (SGD), on four deep RL tasks selected from the ALE (Bellemare et al., 2013) (we use NoFrameSkip-v4 from OpenAI Gym (Brockman et al., 2016)): Beam Rider, Breakout, Qbert, and Space Invaders. Pseudocode is provided for the different optimizers that were used: TDprop in algorithm 5 (chapter 7), Adam in algorithm 4 (chapter 2), and SGD in algorithm 3 (chapter 2). We select these four games based on a random sampling from the original DQN benchmark paper (Mnih et al., 2013). We use n-step expected SARSA (van Seijen et al., 2009), modifying the A2C implementation of Kostrikov (2018), to train each agent. Specifically, in 16 parallel threads we sample 5 transitions using the current policy. We then perform multi-step expected SARSA updates based on the acquired batch of transitions and repeat the sampling process. For the hyperparameter search we sample 50 random hyperparameter sets from the ranges that are summarized in table 7.1. The network architecture is similar to the A2C implementation (Kostrikov, 2018) without the layer for the policy. Finally, the code repository is located at the following url: github.com/joshromoff/tdprop.

Figure 7.1 shows the results of randomly sampling 50 hyperparameter configurations. For the top 25th percentile of hyperparameters TDprop performs as well as, or significantly better than Adam in all four games. However, confirming the theory of theorem 7.6.5, we find that vanilla SGD, under optimal learning rates, performs as well as, or better than Adam, in all games tested while also coming close to TDprop's performance. Our results suggest that while TDProp improves performance by a small, but statistically significant amount under a hyperparameter search in some settings, SGD can as well in other settings.

Figure 7.2 shows a scatter plot of the performance of the algorithms with respect to the learning rate. We find that in certain tasks SGD prefers a considerably larger learning rate than both Adam and TDprop, roughly two orders of magnitude larger at approximately $> 10^{-0.5}$ for Qbert, Breakout, and Beam Rider. However, the optimal learning rate for SGD on Space Invaders was drastically smaller at approximately $10^{-2.5}$. This discrepancy is not seen for both TDprop and Adam, which both tend to have values close to 10^{-3} as the optimal choice across tasks.

We compare the effect of each hyperparameter for TDprop and Adam in Figure 7.3. Specifically, we measure the difference in performance between TDprop and Adam (TDprop - Adam) across tasks and hyperparameters. We find that in most tasks (except for Qbert), TDprop has a better overall coverage of the hyperparameter space, suggesting it improves stability in the non-convex regime.

Hyperparameter	**Range**	**Distribution**
(α) Learning rate (TDprop and Adam)	`[10e-8, 10e-3]`	uniform
(α) Learning rate (SGD)	`[10e-4, 10e-0]`	uniform
(β_2) tracking parameter	`[0, 1]`	uniform
(ϵ) damping parameter	`[10e-8, 10e-1]`	uniform

Table 7.1: The ranges used in sampling hyperparameters

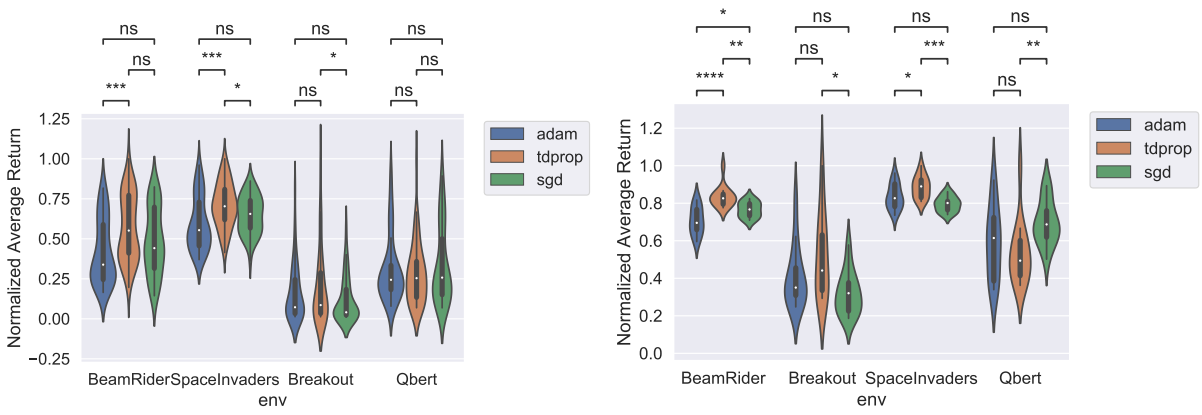

Figure 7.1: The normalized average returns of all hyperparameter configurations within (left) and top 25th percentile (right). Normalization is performed by taking the maximum value for the game and dividing all results by this value. Significance tests are done using Welch's t-test, per recommendations from Colas et al. (2019); Henderson et al. (2017). P-value annotation legend is as follows. ns: $0.05 < p <= 1$; *: $0.01 < p <= 0.05$; **: $0.001 < p <= 0.01$; ***: $0.0001 < p <= 0.001$; ****: $p <= 0.0001$.

All Hyperparameter Samples			
Game	SGD	Adam	TDprop
BeamRider	*778.6 (686.4, 867.7)* †	*673.9 (584.4, 760.4)* †	**907.7 (815.3, 995.3)**
Breakout	12.2 (7.9, 16.0)	16.9 (11.7, 21.7)	20.2 (13.3, 26.5)
SpaceInvaders	*330.6 (314.3, 347.5)* †	302.3 (278.3, 325.1)	**362.3 (343.1, 381.3)**
Qbert	654.3 (519.5, 781.7)	599.2 (483.4, 703.9)	552.3 (444.0, 651.0)
Top 25%			
BeamRider	*1226.2 (1192.4, 1259.7)* †	1131.8 (1069.3, 1192.5)	**1336.2 (1282.2, 1377.5)**
Breakout	33.0 (26.5, 38.9)	42.1 (32.5, 50.0)	53.9 (41.1, 65.1) *
SpaceInvaders	*404.9 (394.3, 415.5)*	*425.0 (407.1, 442.3)*†	**451.4 (435.7, 467.0)**
Qbert	1366.8 (1243.9, 1483.3)*	1157.8 (956.8, 1351.3)	1048.5 (860.6, 1199.5)

Table 7.2: For up to 10M timesteps. Average return with bootstrap confidence intervals in parentheses. Bolded text indicates best based on bootstrap significance test. † indicates runner up by significance testing. If multiple values fall into a tier, denote them by the same marker. For the top 25% of TDprop vs SGD on QBert, the only significant comparison is against SGD (TDProp is significantly worse than SGD, but Adam is not significantly better (or worse) than TDprop or SGD. Conversely for SGD and TDprop on Breakout. This is indicated by *.

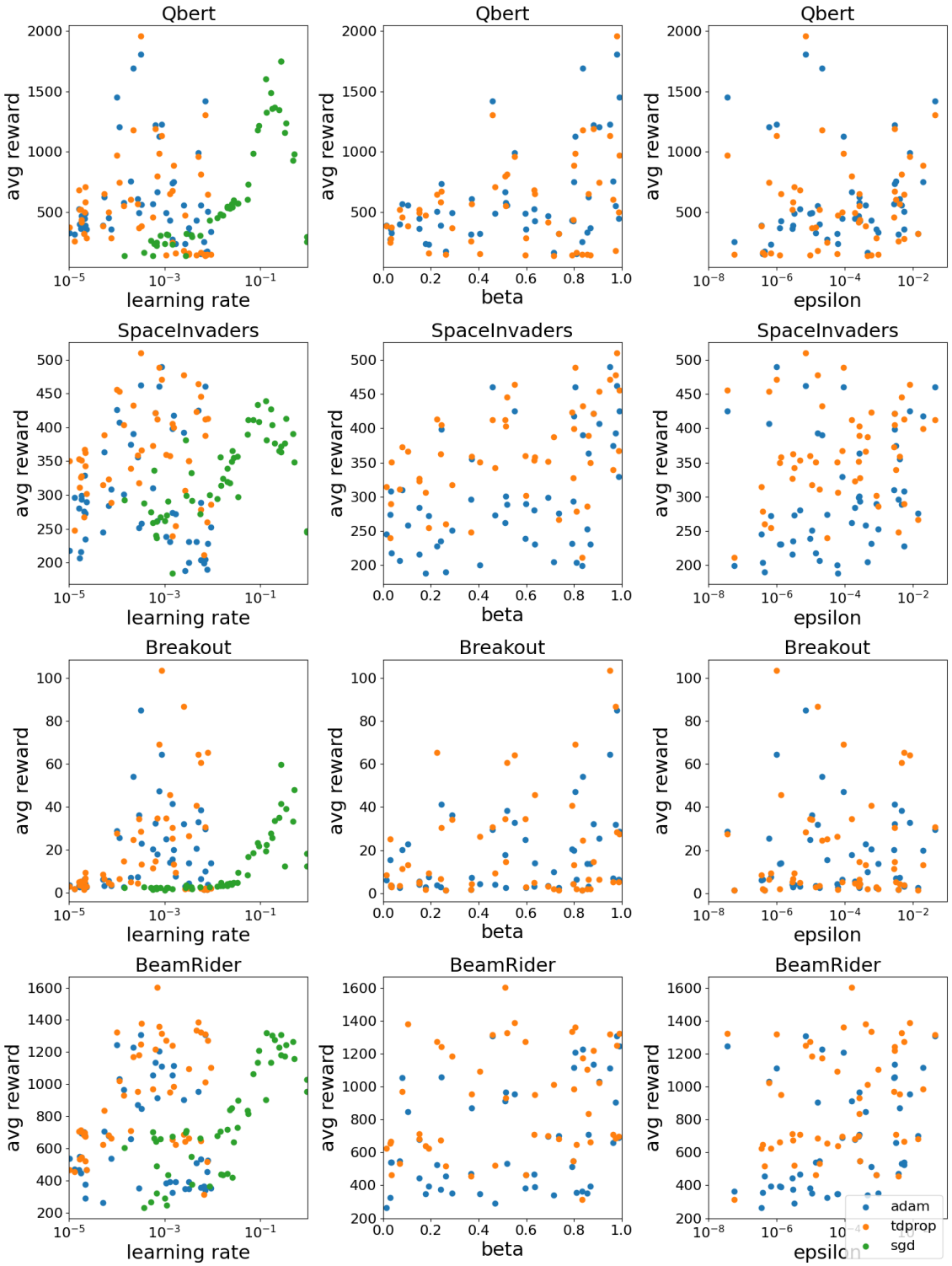

Figure 7.2: Scatter plots of the hyperparameter search on Qbert, Breakout, Space Invaders, and Beam Rider. The y-axis represents the average undiscounted return per episode over 10 million training steps. The x-axis is the learning rate.

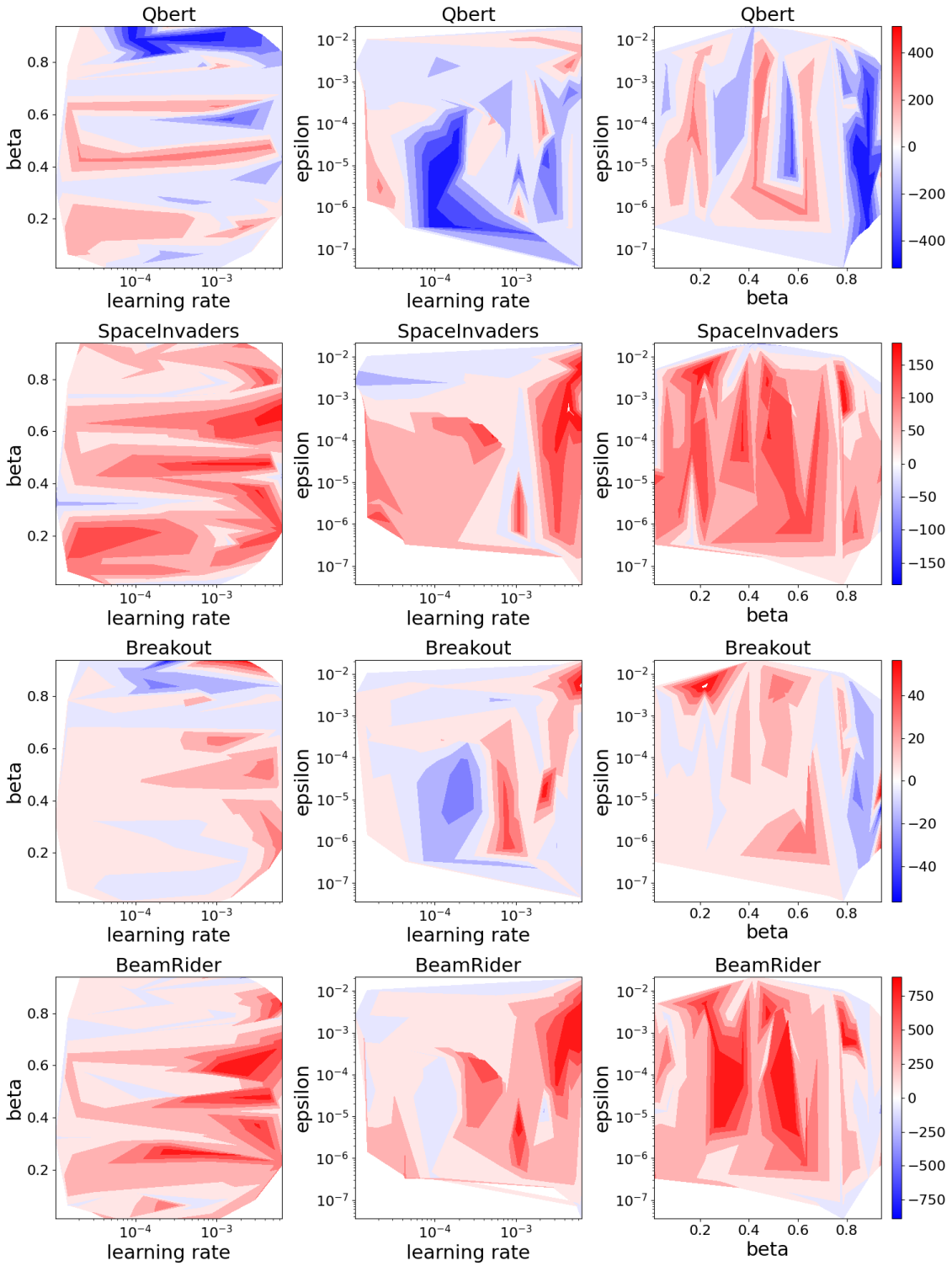

Figure 7.3: Heat-maps of the hyperparameter search on Qbert, Breakout, Space Invaders, and Beam Rider. We measure the performance difference between algorithms (TDprop - Adam), where performance is measured in terms of the average undiscounted return per episode over 10 million training steps. Red areas indicate where TDprop > Adam and blue areas indicate where Adam > TDprop.

7.9 Discussion

In this chapter, we proposed using Jacobi preconditioning for TD learning to adapt a per-parameter learning rate throughout training. We highlighted that in the iterative policy evaluation setting, Jacobi preconditioning, known in this setting as Jacobi matrix splitting (Puterman, 1994), has a faster convergence rate than the non-preconditioned system. While this result was already known in the literature, we extended these results to both the n-step and λ-return settings, proving analogous convergence rate improvements. We note that since both n-step and λ-returns can also be seen as matrix splittings (Bacon, 2018), our corresponding Jacobi splitting can be interpreted as a splitting of a splitting.

Theoretically, we showed that the convergence rate improvement for Jacobi splitting does not necessarily extend to cases where a constant learning rate can be tuned. Empirically, we derived a practical algorithm based off of Jacobi preconditioning that is competitive with state of the art adaptive optimizers in the deep RL literature. However, we note that consistent with the theory, once the global learning rate has been tuned, all of the optimizers that were tested performed similarly. Finally, due to the similar optimal performance of TDprop, SGD, and Adam, we did not perform additional experiments with decompositions such as TD(Δ).

An interesting avenue of future work would be to explore the different interpretations of Jacobi matrix splitting, which notably can also be understood as prescribing a per state learning rate or a per state n-step return. As we mentioned in section 7.5, in the tabular case, Jacobi preconditioning can be interpreted as a per state learning rate based on the probability of self-looping. The extension to function approximation is not trivial, as self-looping in the function approximation setting is ill-defined, since the agent almost never revisits the exact same state. Notably, this was our initial approach for the project, however, once we discovered the use of Jacobi preconditioning in the stochastic approximation literature, the project switched from learning a per-state learning to tracking a per-parameter learning rate. While similar in the tabular case, importantly, the latter is much better defined in the function approximation setting. Nevertheless, a model-based approach that learns an estimate for the probability of self-looping and uses this directly as a per-state learning rate would be an interesting research direction. However, we would expect to at best achieve similar results to TDprop, since the theory presented in this chapter implies that once the learning rate has been tuned, a per state learning rate based on Jacobi preconditioning is not guaranteed to have better performance.

Alternatively, another potential research direction could be to explore implementing Jacobi matrix splitting in terms of random stopping times (Wessels, 1977; Bacon, 2018). Instead of

a per-state learning rate, in this case, Jacobi splitting can be interpreted as a per-state n-step return based on self-looping. Specifically, n varies from state to state and from trajectory to trajectory. As long as the agent remains in the same state as the reference state s_t, the current n is increased, and another sample is used in the current return. Recently, there have been some works that vary the amount of bootstrapping on a state by state basis based on meta-gradients (Xu et al., 2018). The random stopping times prescribed by Jacobi splitting could be an interesting alternative to such approaches.

Chapter 8

Conclusion

We have presented several different methods for improving the sample efficiency and brittleness of deep RL methods through decomposing the value estimation problem into simple components. The improvement of the proposed methods can be understood as breaking the equivalence theorem from chapter 6. Specifically, from this theorem we were able to understand exactly what makes these decompositions similar to non-decomposed systems and more importantly, what makes them different.

8.1 Summary of Contributions

Our first method, that we presented in chapter 3, decomposed the reward function into components to be solved by separate value functions that were subsequently aggregated together. Notably, we highlighted that optimizing each individual value function to be locally optimal may result in undesirable behaviour. For example, if training two value functions to collect two separate objects, the aggregation may result in neither object being collected. Moreover, as predicted by the Bellman equivalence theorem, optimizing towards the globally optimal policy will lead to equivalent performance to non-decomposed approaches. Instead, we propose to optimize each of the decomposed value functions towards the random policy. By using this objective we were able to use a smaller state representation to train each decomposed value function. Specifically, we were able to train value functions using tabular methods, which greatly increased sample efficiency by removing the instabilities associated with function approximation.

In chapters 4 and 5, we explored decompositions based on the discount factor. The discount factor, which controls the weighting of future rewards, is a sensitive hyperparameter in deep RL. Value functions based on small discount factors have faster convergence rates, however, in most deep RL settings we use larger discount factors to capture long term

dependencies. Thus, we designed a novel Bellman-like equation that bootstraps the learning of value functions based on large discount factors with value functions with smaller discounts. We showed that improved sample efficiency can be achieved by breaking the equivalence with standard TD methods by optimizing hyperparameters, such as the n-step λ-return or the learning rate for each time scale.

Moreover, in chapter 4, we further explored this discount-based decomposition, where the initial value function uses a discount factor of 0. In this scenario, the decomposition can be interpreted as using a reward estimator as a replacement for the empirically sampled reward in the TD update rule. We found that under extremely noisy reward functions, the reward predictor greatly improved training stability which lead to improved sample efficiency.

Finally, in chapter 7, we proposed a method that can break the equivalence between decomposed and non-decomposed methods by adapting the learning rate. Specifically, we proposed to use Jacobi preconditioning as an adaptive optimizer for TD learning, which effectively tracks a per-parameter learning rate. Notably, Jacobi preconditioning has the interesting interpretation in the tabular case as using a per state learning rate that is proportional to the probability of remaining in the same state. We extended Jacobi preconditioning to both the multi-step TD case, as well as with function approximation. Ultimately, we found that both theoretically and empirically, it performs similarly to a well tuned SGD.

8.2 Perspective

There are several theoretical and empirical limitations to the results found in this thesis. Theoretically, most of the theory we presented is limited to the case of linear function approximation, with a large amount further limited to the tabular setting. With every year that passes, however, our ability analyze more complex function approximators increases, with their recently being some notable theoretical results of RL with non-linear function approximation (Chen et al., 2019). That being said, developing methods that are rooted in theoretically sound principles in the linear case, and then extending them to the most powerful deep RL technique available, will most likely remain the popular approach for most deep RL researchers going forward.

Empirically, we limited the scope of the projects presented in this thesis to relatively simple tasks. While pixel-based experiments in the ALE are certainly a more realistic setting than the simple toy experiments than comprised RL research a decade or two ago (e.g., Cartpole or Mountain car (Sutton and Barto, 2018)), there is still a large gap between our simulations and real life applications. Recently, several more realistic 3D environments have been proposed using modern day video game engines (Juliani et al., 2018), they present

an interesting challenge for deep RL by adding more visually realistic scenarios as well as typically adding some layer of partial observability.

Despite the limitations, we have seen that hyperparameters including the learning rate, discount factor, and model structure, play a critical role in both the brittleness and sample efficiency of deep RL systems. By decomposing TD methods, each of the individual components can be trained with hyperparameters that are more optimal for their respective decomposition. We hope that our works will help inspire further decompositions that could potentially enable deep RL to have the proper scaling to enter into mainstream products.

Bibliography

Pieter Abbeel and Andrew Y. Ng. Apprenticeship learning via inverse reinforcement learning. In *Proceedings of the Twenty-First International Conference on Machine learning*, 2004.

Mahmoud Assran, Joshua Romoff, Nicolas Ballas, Joelle Pineau, and Michael Rabbat. Gossip-based actor-learner architectures for deep reinforcement learning. In *Advances in Neural Information Processing Systems*, 2019.

Pierre-Luc Bacon. *Temporal Representation Learning*. PhD thesis, McGill University, 2018.

Pierre-Luc Bacon, Jean Harb, and Doina Precup. The option-critic architecture. In *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

Leemon Baird. Residual algorithms: Reinforcement learning with function approximation. In *Machine Learning*, pages 30–37. Elsevier, 1995.

Stefan Banach. Sur les opérations dans les ensembles abstraits et leur application aux équations intégrales. *Fund. math*, 3(1):133–181, 1922.

André Barreto, Will Dabney, Rémi Munos, Jonathan J. Hunt, Tom Schaul, Hado P. van Hasselt, and David Silver. Successor features for transfer in reinforcement learning. In *Advances in Neural Information Processing Systems*, 2017.

Andrew G. Barto and Sridhar Mahadevan. Recent advances in hierarchical reinforcement learning. *Discrete event dynamic systems*, 13(1-2):41–77, 2003.

Marc G. Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, 2013.

Marc G. Bellemare, Sriram Srinivasan, Georg Ostrovski, Tom Schaul, David Saxton, and Remi Munos. Unifying count-based exploration and intrinsic motivation. In *Advances in Neural Information Processing Systems*, 2016.

Richard Bellman. A markovian decision process. *Journal of Mathematics and Mechanics*, pages 679–684, 1957.

Albert Benveniste, Michel Métivier, and Pierre Priouret. *Adaptive algorithms and stochastic approximations*, volume 22. Springer Science & Business Media, 2012.

Dimitri P Bertsekas and John N Tsitsiklis. Neuro-dynamic programming: an overview. In *Proceedings of the Thirty-Fourth IEEE Conference on Decision and Control*, volume 1, pages 560–564. IEEE Publ. Piscataway, NJ, 1995.

Christopher M. Bishop. *Pattern recognition and machine learning*. springer, 2006.

Justin A Boyan. Least-squares temporal difference learning. In *Proceedings of the Sixteenth International Conference on Machine Learning*, pages 49–56. Citeseer, 1999.

Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. OpenAI Gym. *arXiv preprint arXiv:1606.01540*, 2016.

Shuhang Chen, Adithya M Devraj, Ana Bušić, and Sean Meyn. Zap q-learning with nonlinear function approximation. *arXiv preprint arXiv:1910.05405*, 2019.

Erhan Cinlar. *Introduction to stochastic processes*. Courier Corporation, 2013.

Cédric Colas, Olivier Sigaud, and Pierre-Yves Oudeyer. How many random seeds? statistical power analysis in deep reinforcement learning experiments. *1810.025251806.08295*, 2018.

Cédric Colas, Olivier Sigaud, and Pierre-Yves Oudeyer. A hitchhiker's guide to statistical comparisons of reinforcement learning algorithms. *arXiv preprint arXiv:1904.06979*, 2019.

William Dabney. *Adaptive step-sizes for reinforcement learning*. PhD thesis, University of Massachusetts Amherst, 2014.

William Dabney and Andrew G. Barto. Adaptive step-size for online temporal difference learning. In *Twenty-Sixth AAAI Conference on Artificial Intelligence*, 2012.

Felix Dangel, Frederik Kunstner, and Philipp Hennig. Backpack: Packing more into backprop. *arXiv preprint arXiv:1912.10985*, 2019.

Abhishek Das, Théophile Gervet, Joshua Romoff, Dhruv Batra, Devi Parikh, Mike Rabbat, and Joelle Pineau. Tarmac: Targeted multi-agent communication. In *Proceedings of the Thirty Sixth International Conference on Machine Learning*, 2019.

Yann Dauphin, Harm De Vries, and Yoshua Bengio. Equilibrated adaptive learning rates for non-convex optimization. In *Advances in Neural Information Processing Systems*, 2015.

Adithya M Devraj and Sean Meyn. Zap q-learning. In *Advances in Neural Information Processing Systems*, 2017.

Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, and Yuhuai Wu. OpenAI Baselines. `https://github.com/openai/baselines`, 2017.

Thomas G Dietterich. Hierarchical reinforcement learning with the maxq value function decomposition. *Journal of Artificial Intelligence Research*, 13:227–303, 2000.

John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul): 2121–2159, 2011.

Eyal Even-Dar and Yishay Mansour. Learning rates for q-learning. *Journal of Machine Learning Research*, 5(Dec):1–25, 2003.

Tom Everitt, Victoria Krakovna, Laurent Orseau, Marcus Hutter, and Shane Legg. Reinforcement learning with a corrupted reward channel. *arXiv preprint arXiv:1705.08417*, 2017.

William Fedus, Mehdi Fatemi, Yoshua Bengio, Marc G. Bellemare, and Hugo Larochelle. Hyperbolic discounting and learning over multiple horizons. *arXiv preprint arXiv:1902.06865*, 2019.

Eugene A. Feinberg and Adam Shwartz. Markov decision models with weighted discounted criteria. *Mathematics of Operations Research*, 19(1):152–168, 1994.

Vladimir Feinberg, Alvin Wan, Ion Stoica, Michael I. Jordan, Joseph E. Gonzalez, and Sergey Levine. Model-based value estimation for efficient model-free reinforcement learning. *arXiv preprint arXiv:1803.00101*, 2018.

Meire Fortunato, Mohammad Gheshlaghi Azar, Bilal Piot, Jacob Menick, Ian Osband, Alex Graves, Vlad Mnih, Remi Munos, Demis Hassabis, Olivier Pietquin, Charles Blundell, and Shane Legg. Noisy networks for exploration. *arXiv preprint arXiv:1706.10295*, 2017.

Vincent François-Lavet, Raphael Fonteneau, and Damien Ernst. How to discount deep reinforcement learning: Towards new dynamic strategies. *arXiv preprint arXiv:1512.02011*, 2015.

Vincent François-Lavet, Yoshua Bengio, Doina Precup, and Joelle Pineau. Combined reinforcement learning via abstract representations. *arXiv preprint arXiv:1809.04506*, 2018.

Izrail Gelfand. Normierte ringe. *Rec. Math. [Mat. Sbornik] N.S.*, 9(1):3–24, 1941.

Arash Givchi and Maziar Palhang. Quasi newton temporal difference learning. In *Asian Conference on Machine Learning*, 2015.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

Anne Greenbaum. *Iterative methods for solving linear systems*, volume 17. Siam, 1997.

Anna Harutyunyan, Marc G. Bellemare, Tom Stepleton, and Rémi Munos. Q(λ) with off-policy corrections. *arXiv preprint arXiv:1602.04951*, 2016.

Mikael Henaff, William F. Whitney, and Yann LeCun. Model-based planning in discrete action spaces. *arXiv preprint arXiv:1705.07177*, 2017.

Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. Deep reinforcement learning that matters. *arXiv preprint arXiv:1709.06560*, 2017.

Peter Henderson, Joshua Romoff, and Joelle Pineau. Where did my optimum go?: An empirical analysis of gradient descent optimization in policy gradient methods. *arXiv preprint arXiv:1810.02525*, 2018.

Peter Henderson, Jieru Hu, Joshua Romoff, Emma Brunskill, Dan Jurafsky, and Joelle Pineau. Towards the systematic reporting of the energy and carbon footprints of machine learning. *arXiv preprint arXiv:2002.05651*, 2020.

Bernhard Hengst. Discovering hierarchy in reinforcement learning with hexq. In *ICML*, volume 2, pages 243–250, 2002.

Ronald A Howard. *Dynamic programming and markov processes.* John Wiley, 1960.

David Hubel and Torsten Wiesel. Receptive fields and functional architecture of monkey striate cortex. *Journal of Physiology*, 195:215–243, 1968.

Max Jaderberg, Volodymyr Mnih, Wojciech Marian Czarnecki, Tom Schaul, Joel Z. Leibo, David Silver, and Koray Kavukcuoglu. Reinforcement learning with unsupervised auxiliary tasks. *arXiv preprint arXiv:1611.05397*, 2016.

Zhengyao Jiang, Dixing Xu, and Jinjun Liang. A deep reinforcement learning framework for the financial portfolio management problem. *arXiv preprint arXiv:1706.10059*, 2017.

Arthur Juliani, Vincent-Pierre Berges, Esh Vckay, Yuan Gao, Hunter Henry, Marwan Mattar, and Danny Lange. Unity: A general platform for intelligent agents. *arXiv preprint arXiv:1809.02627*, 2018.

Lukasz Kaiser, Mohammad Babaeizadeh, Piotr Milos, Blazej Osinski, Roy H. Campbell, Konrad Czechowski, Dumitru Erhan, Chelsea Finn, Piotr Kozakowski, Sergey Levine, Ryan Sepassi, George Tucker, and Henryk Michalewski. Model-based reinforcement learning for atari. *arXiv preprint arXiv:1903.00374*, 2019.

Alex Kearney, Vivek Veeriah, Jaden Travnik, Patrick M Pilarski, and Richard S. Sutton. Learning feature relevance through step size adaptation in temporal-difference learning. *arXiv preprint arXiv:1903.03252*, 2019.

Michael Kearns and Satinder Singh. Near-optimal reinforcement learning in polynomial time. *Machine learning*, 49(2-3):209–232, 2002.

Michael J Kearns and Satinder P Singh. Bias-variance error bounds for temporal difference updates. In *Proceedings of the Thirteenth Annual Conference on Computational Learning Theory*, 2000.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

W. Bradley Knox and Peter Stone. Reinforcement learning from simultaneous human and mdp reward. In *Proceedings of the Eleventh International Conference on Autonomous Agents and Multiagent Systems*, 2012.

George Konidaris, Scott Niekum, and Philip S. Thomas. TD(γ): Re-evaluating complex backups in temporal difference learning. In *Advances in Neural Information Processing Systems*, 2011.

Ilya Kostrikov. Pytorch implementations of reinforcement learning algorithms, 2018.

Tejas D. Kulkarni, Karthik Narasimhan, Ardavan Saeedi, and Josh Tenenbaum. Hierarchical deep reinforcement learning: Integrating temporal abstraction and intrinsic motivation. In *Advances in Neural Information Processing Systems*, 2016.

Romain Laroche and Merwan Barlier. Transfer reinforcement learning with shared dynamics. In *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

Romain Laroche, Mehdi Fatemi, Joshua Romoff, and Harm van Seijen. Multi-advisor reinforcement learning. *arXiv preprint arXiv:1704.00756*, 2017.

Yann LeCun, Bernhard Boser, John S. Denker, Donnie Henderson, Richard E. Howard, Wayne Hubbard, and Lawrence D. Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551, 1989.

Yann LeCun, Léon Bottou, Genevieve B. Orr, and Klaus-Robert Müller. Efficient backprop. In *Neural networks: Tricks of the trade*, pages 9–48. Springer, 2012.

Jae Won Lee, Jonghun Park, O. Jangmin, Jongwoo Lee, and Euyseok Hong. A multiagent approach to q-learning for daily stock trading. *IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans*, 37(6):864–877, 2007.

Yitao Liang, Marlos C Machado, Erik Talvitie, and Michael Bowling. State of the art control of atari games using shallow reinforcement learning. *arXiv preprint arXiv:1512.01563*, 2015.

Long-Ji Lin. *Reinforcement Learning for Robots Using Neural Networks*. PhD thesis, Carnegie Mellon University, 1992.

James Martens. New insights and perspectives on the natural gradient method. *arXiv preprint arXiv:1412.1193*, 2014.

Ishai Menache, Shie Mannor, and Nahum Shimkin. Q-cut—dynamic discovery of sub-goals in reinforcement learning. In *European Conference on Machine Learning*, 2002.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *Proceedings of the Thirty-Third International Conference on Machine Learning*, pages 1928–1937, 2016.

Vinod Nair and Geoffrey E. Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of The Twenty-Seventh International Conference on Machine Learning*, 2010.

Andrew Y. Ng and Stuart J. Russell. Algorithms for inverse reinforcement learning. In *Proceedings of the Seventeenth International Conference on Machine Learning*, 2000.

Chuong V Nguyen, Shahram Izadi, and David Lovell. Modeling kinect sensor noise for improved 3d reconstruction and tracking. In *International Conference on 3D Imaging, Modeling, Processing, Visualization and Transmission*, 2012.

OpenAI. Openai five. `https://blog.openai.com/openai-five/`, 2018.

Ian Osband, Benjamin van Roy, Daniel J Russo, and Zheng Wen. Deep exploration via randomized value functions. *Journal of Machine Learning Research*, 20(124):1–62, 2019.

Yangchen Pan, Adam White, and Martha White. Accelerated gradient temporal difference learning. In *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, 2019.

Theodore J Perkins and Doina Precup. A convergent form of approximate policy iteration. In *Advances in Neural Information Processing Systems*, 2003.

Boris T. Polyak and Anatoli B. Juditsky. Acceleration of stochastic approximation by averaging. *Journal on Control and Optimization*, 30(4):838–855, 1992.

Lee C. Potter, Emre Ertin, Jason T. Parker, and Müjdat Cetin. Sparsity and compressed sensing in radar imaging. *Proceedings of the IEEE*, 98(6):1006–1020, 2010.

Danil V. Prokhorov and Donald C. Wunsch. Adaptive critic designs. *IEEE transactions on Neural Networks*, 8(5):997–1007, 1997.

Martin L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons, Inc., 1st edition, 1994.

Sébastien Racanière, Theophane Weber, David Reichert, Lars Buesing, Arthur Guez, Danilo Jimenez Rezende, Adrià Puigdomènech Badia, Oriol Vinyals, Nicolas Heess, Yujia Li, Razvan Pascanu, Peter Battaglia, Demis Hassabis, David Silver, and Daan Wierstra. Imagination-augmented agents for deep reinforcement learning. In *Advances in Neural Information Processing Systems*, 2017.

Chris Reinke, Eiji Uchibe, and Kenji Doya. Average reward optimization with multiple discounting reinforcement learners. In *International Conference on Neural Information Processing*, 2017.

Herbert Robbins and Sutton Monro. A stochastic approximation method. *The annals of mathematical statistics*, pages 400–407, 1951.

Joshua Romoff, Peter Henderson, Alexandre Piché, Vincent François-Lavet, and Joelle Pineau. Reward estimation for variance reduction in deep reinforcement learning. In *Proceedings of the Second Conference on Robot Learning*, 2018a.

Joshua Romoff, Alexandre Piche, Peter Henderson, Vincent Francois-Lavet, and Joelle Pineau. Reward estimation for variance reduction in deep reinforcement learning, 2018b. URL `https://openreview.net/forum?id=r1vcHYJvM`.

Joshua Romoff, Peter Henderson, Ahmed Touati, Emma Brunskill, Joelle Pineau, and Yann Ollivier. Separating value functions across time-scales. In *Proceedings of the Thirty Sixth International Conference on Machine Learning*, 2019.

Joshua Romoff, Peter Henderson, David Kanaa, Emmanuel Bengio, Ahmed Touati, Pierre-Luc Bacon, and Joelle Pineau. TDprop: Does Jacobi preconditioning help temporal difference learning? *arXiv preprint arXiv:2007.02786*, 2020.

Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

Gavin A. Rummery and Mahesan Niranjan. *On-line Q-learning using connectionist systems*, volume 37. University of Cambridge, Department of Engineering Cambridge, UK, 1994.

Stuart J Russell and Andrew Zimdars. Q-decomposition for reinforcement learning agents. In *Proceedings of the Twentieth International Conference on Machine Learning*, 2003.

Tom Schaul, Sixin Zhang, and Yann LeCun. No more pesky learning rates. In *Proceedings of the Thirtieth International Conference on Machine Learning*, 2013.

Tom Schaul, Daniel Horgan, Karol Gregor, and David Silver. Universal value function approximators. In *Proceedings of the Thirty-Second International conference on machine learning*, 2015.

Ralf Schoknecht and Artur Merke. TD(0) converges provably faster than the residual gradient algorithm. In *Proceedings of the Twentieth International Conference on Machine Learning*, 2003.

Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, Timothy Lillicrap, and David Silver. Mastering atari, go, chess and shogi by planning with a learned model. *arXiv preprint arXiv:1911.08265*, 2019.

John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438*, 2015.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Craig Sherstan, James MacGlashan, and Patrick M Pilarski. Generalizing value estimation over timescale. *FAIM Workshop on Prediction and Generative Modeling in Reinforcement Learning*, 2018.

David Silver, Aja Huang, Christopher J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016a.

David Silver, Hado P. van Hasselt, Matteo Hessel, Tom Schaul, Arthur Guez, Tim Harley, Gabriel Dulac-Arnold, David P. Reichert, Neil C. Rabinowitz, André Barreto, and Thomas Degris. The predictron: End-to-end learning and planning. *arXiv preprint arXiv:1612.08810*, 2016b.

Satinder Singh, Tommi Jaakkola, Michael L. Littman, and Csaba Szepesvári. Convergence results for single-step on-policy reinforcement-learning algorithms. *Machine learning*, 38 (3):287–308, 2000.

Nathan Sprague and Dana Ballard. Multiple-goal reinforcement learning with modular SARSA(0). In *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence*, 2003.

Tao Sun, Han Shen, Tianyi Chen, and Dongsheng Li. Adaptive temporal difference learning with linear function approximation. *arXiv preprint arXiv:2002.08537*, 2020.

Richard S. Sutton. Learning to predict by the methods of temporal differences. *Machine learning*, 3(1):9–44, 1988.

Richard S. Sutton. Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *Proceedings of the Seventh International Conference on Machine Learning*, pages 216–224, 1990.

Richard S. Sutton. Dyna, an integrated architecture for learning, planning, and reacting. *ACM Sigart Bulletin*, 2(4):160–163, 1991.

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

Richard S. Sutton. Generalization in reinforcement learning: Successful examples using sparse coarse coding. In *Advances in Neural Information Processing Systems*, 1996.

Richard S. Sutton and Andrew G. Barto. *Introduction to reinforcement learning*. MIT press Cambridge, second edition, 2018.

Richard S. Sutton, Doina Precup, and Satinder Singh. Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, 112 (1-2):181–211, 1999.

Richard S. Sutton, David McAllester, Satinder Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Advances in Neural Information Processing Systems*, 2000.

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 *Proceedings of the International Conference on Autonomous Agents and Multiagent Systems*, 2011.

Erik Talvitie. Learning the reward function for a misspecified model. *arXiv preprint arXiv:1801.09624*, 2018.

Matthew E Taylor and Peter Stone. Transfer learning for reinforcement learning domains: A survey. *Journal of Machine Learning Research*, 10(7), 2009.

Edward L Thorndike. Animal intelligence: an experimental study of the associative processes in animals. *The Psychological Review: Monograph Supplements*, 2(4):i, 1898.

Tim Tieleman and Goeffrey E. Hinton. Lecture 6.5—RmsProp: Divide the gradient by a running average of its recent magnitude. Coursera: Neural Networks for Machine Learning, 2012.

Ahmed Touati, Harsh Satija, Joshua Romoff, Joelle Pineau, and Pascal Vincent. Randomized value functions via multiplicative normalizing flows. *arXiv preprint arXiv:1806.02315*, 2018.

John N Tsitsiklis and Benjamin van Roy. Analysis of temporal-diffference learning with function approximation. In *Advances in Neural Information Processing Systems*, 1997.

Hado P. van Hasselt, Arthur Guez, Matteo Hessel, Volodymyr Mnih, and David Silver. Learning values across many orders of magnitude. In *Advances in Neural Information Processing Systems*, 2016.

Harm van Seijen and Richard S. Sutton. Efficient planning in MDPs by small backups. In *Proceedings of the International Conference on Machine Learning*, 2013.

Harm van Seijen, Hado P. van Hasselt, Shimon Whiteson, and Marco Wiering. A theoretical and empirical analysis of expected SARSA. In *IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning*, 2009.

Harm van Seijen, Mehdi Fatemi, Joshua Romoff, and Romain Laroche. Separation of concerns in reinforcement learning. *arXiv preprint arXiv:1612.05159*, 2016.

Harm van Seijen, Mehdi Fatemi, Joshua Romoff, Romain Laroche, Tavian Barnes, and Jeffrey Tsang. Hybrid reward architecture for reinforcement learning. In *Advances in Neural Information Processing Systems*, 2017.

Richard S. Varga. *Matrix Iterative Analysis*. Prentice-Hall, 1962.

Oriol Vinyals, Igor Babuschkin, Wojciech Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David Choi, Richard Powell, Timo Ewalds, Petko Georgiev, Junhyuk Oh, Dan Horgan, Manuel Kroiss, Ivo Danihelka, Aja Huang, Laurent Sifre, Trevor Cai, John Agapiou, Max Jaderberg, and David Silver. Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature*, 575(7782):350–354, 2019.

Christopher John Cornish Hellaby Watkins. *Learning from delayed rewards*. PhD thesis, King's College, Cambridge, 1989.

J. Wessels. Stopping times and markov programming. In *Transactions of the Seventh Prague Conference on Information Theory, Statistical Decision Functions, Random Processes and of the 1974 European Meeting of Statisticians*, 1977.

Martha White. Unifying task specification in reinforcement learning. In *Proceedings of the Thirty-Fourth International Conference on Machine Learning*, 2017.

Bernard Widrow and Marcian E. Hoff. Adaptive switching circuits. Technical report, Stanford Electronics Labs, 1960.

Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8(3):229–256, 1992.

David Wingate. *Solving large MDPs quickly with partitioned value iteration*. PhD thesis, Brigham Young University, 2004.

David Wingate and Kevin D Seppi. Efficient value iteration using partitioned models. In *Proceedings of the Second International Conference on Machine Learning and Applications*, 2003.

Zhongwen Xu, Hado P. van Hasselt, and David Silver. Meta-gradient reinforcement learning. *arXiv preprint arXiv:1805.09801*, 2018.

Hengshuai Yao and Zhi-Qiang Liu. Preconditioned temporal difference learning. In *Proceedings of the Twenty-Fifth International Conference on Machine learning*, 2008.

Hengshuai Yao, Shalabh Bhatnagar, and Csaba Szepesvári. Temporal difference learning by direct preconditioning. *Multidisciplinary Symposium on Reinforcement Learning*, 2009.

Chao Yu, Jiming Liu, and Shamim Nemati. Reinforcement learning in healthcare: a survey. *arXiv preprint arXiv:1908.08796*, 2019.