Generalization, optimization, diverse generation: insights and advances in the use of bootstrapping in deep neural networks

Emmanuel Bengio

Computer Science
McGill University, Montreal

April, 2022

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

© Emmanuel Bengio, 2021.

Abstract

This thesis investigates the use of bootstrapping in Temporal Difference (TD) learning, a central mechanism in reinforcement learning (RL), when applied to deep neural networks. I first investigate generalization in deep learning and deep RL, and show that it can be understood through the lens of gradient interference. This perspective shows a need to adapt the tools of deep RL that are naively imported from modern supervised learning. Using this insight, I propose a novel optimization method for TD based on compensating for the staleness induced by modern optimizers. I then propose a novel generative modeling method based on bootstrapping. Finally, I propose a novel representation learning method to jointly learn to act in and understand the world, again addressing the issue of generalization in deep reinforcement learning.

Résumé

Cette thèse étudie l'utilisation du *bootstrapping* en apprentissage par la méthode des Différences Temporelles (TD), un mécanisme central en apprentissage par renforcement (RL), tel qu'appliqué aux réseaux de neurones profonds. J'investigue premièrement la généralisation en apprentissage profond ainsi qu'en apprentissage par renforcement profond (*deep RL*), montrant que celle-ci peut être comprise via une perspective d'interférence des gradients. Cette perspective démontre un besoin d'adapter les outils du *deep RL*, alors que ceux-ci sont naïvement importés de l'apprentissage supervisé moderne. De par cette réalisation, je propose une nouvelle méthode d'optimisation pour l'apprentissage par TD, basée sur la compensation d'un biais d'ancienneté (*staleness*) induit par les optimisateurs modernes. Puis, je propose un nouvel algorithme génératif basé sur le *bootstrapping*. Finalement, je propose une nouvelle méthode d'apprentissage de représentations, qui apprend simultanément à agir et à comprendre le monde, encore une fois dans le but d'adresser les problèmes de généralisation en apprentissage par renforcement profond.

Contributions to Original Knowledge

This thesis contributes to our comprehension of the use of bootstrapping and deep neural networks from a number of point of views. Specifically, it proposes:

1. A perspective on generalization based around a functional definition of its opposite, memorization, which shows that:
    - memorization is a feature, not a bug: deep neural networks rely on individual examples to map the space of general patterns and in doing so generalize;
    - when using bootstrapping for Temporal Difference (TD) learning, generalization (or lack thereof) behaviours can be partially explained by interference, i.e. how learning about a new example affects what has been learned before.
2. An optimization algorithm for methods using bootstrapping, in particular TD learning, which:
    - is based on a notion of *staleness* in momentum, whose existence is empirically verified;
    - provides a first-order correction which speeds up convergence of TD learning in the evaluation setting.
3. A framework designed for the sampling of discrete objects, based on the concept of diversity through reward-proportional sampling. This framework:
    - is based on a notion of flows and flow-consistency, which is shown to achieve the desired reward-proportional sampling;
    - can be materialized through a bootstrapping-based objective for function approximators;
    - is shown empirically to be able to diversely explore the space of drug-like molecules while finding high-reward candidates.

4. A representation learning framework based on jointly learning to represent the world and act in it, which leverages the *independence* of the factors that an agent can control. This framework is shown to be able to recover such factors in simple settings.

Contributions of Authors

- Chapter 1 provides a high-level overview of the field of Artificial Intelligence, the social and scientific context in which this thesis is written, as well as central hypotheses of the thesis. Chapter 2 provides the technical foundations of this thesis. Both are original material.
- Chapter 3 is built around two papers, Arpit et al. (2017) and Bengio, Pineau, and Precup (2020a).
  - The first is a collaboration between a number of colleagues (with the four first authors having equal contribution): Devansh Arpit, Stanisław Jastrzębski, Nicolas Ballas, David Krueger, myself, Maxinder S Kanwal, Tegan Maharaj, Asja Fischer, Aaron Courville, Yoshua Bengio, and Simon Lacoste-Julien. While we collectively wrote an entire paper to which I have contributed to here and there, there is a particular subsection and its corresponding set of experiments which I have produced. Section 3.1 thus highlights and reuses the material from my particular contribution to this paper, accompanied by a summary of its other results as well as the context in which this paper was written.
  - Section 3.2 is based on Bengio, Pineau, and Precup (2020a) and reproduces its material for the most part, with some adjustment and extra contextualizing. The code and experiments were written by myself, while the paper was written with the help of my supervisors Joelle Pineau and Doina Precup.
- Chapter 4 is based on Bengio, Pineau, and Precup (2020b), and also reproduces the original material fairly closely, with the code and experiments produced by myself, and again writing done with the help of Joelle Pineau and Doina Precup.
- Chapter 5 is based on Bengio, Jain, Korablyov, Precup, and Bengio (2021), which emerged through the Mila Molecule Discovery project, led by Maksym Korablyov. Specifically, Yoshua Bengio and myself contributed to the original idea, and wrote most sections of the paper. Yoshua Bengio wrote the proofs of Propositions 1-3, and

I the proof of Proposition 4. I wrote the code and ran experiments for the single-round reward-fitting setting. Moksh Jain wrote the code and ran experiments for the multi-round setting and wrote the corresponding results section of the paper. Maksym Korablyov wrote the biochemical framework upon which the molecule experiments are built, assisted in debugging and running experiments, implemented mode-counting routines, and wrote the biochemical details of the paper. Maksym Korablyov, Doina Precup and Yoshua Bengio provided supervision for the project. All authors contributed to proofreading and editing the paper.

- Chapter 6 is based on Bengio, Thomas, Pineau, Precup, and Bengio (2017), Thomas, Pondard, Bengio, Sarfati, Beaudoin, Meurs, Pineau, Precup, and Bengio (2017), and Thomas, Bengio, Fedus, Pondard, Beaudoin, Larochelle, Pineau, Precup, and Bengio (2018), but the material reproduced in this thesis follows the second paper more closely.
  - Bengio et al. (2017) is the first paper of the series, and stems from an original idea from Yoshua Bengio, which I implemented and produced results for. Valentin Thomas then joined the effort, producing results for the second half of the paper.
  - Thomas et al. (2017) and Thomas et al. (2018) are attempts to follow up the original work by extending it to the continuous domain. Attribution is fairly spread, but there is a core group, Valentin Thomas, Jules Pondard, William Fedus, and myself, who worked very closely on finding ways to make the ideas in this project come to reality (and to whom equal contribution is credited in those papers). Marc Sarfati contributed to some of the experimental effort, while other authors acted as supervisors and gave some help in writing the papers.

Acknowledgements

This thesis would not have been the same without the freedom and support that my supervisors have given me, and I'm very thankful to Joelle Pineau and Doina Precup. Thank you for always pushing me to think like a scientist.

I thank my collaborators and coauthors, Aaron Courville, Andrei Nica, Asja Fischer, Danny Tarlow, David Krueger, Devansh Arpit, Hugo Larochelle, Jackie CK Cheung, Joshua Romoff, Jules Pondard, Leo Long, Liam Fedus, Maksym Korablyov, Marc Sarfati, Marie-Jean Meurs, Maxinder S. Kanwal, Moksh Jain, Nicolas Ballas, Paul François, Philippe Beaudoin, Ryan Lowe, Simon Lacoste-Julien, Stanisław Jastrzębski, Tegan Maharaj, Thomas J. Rademaker, Valentin Thomas, and Yoshua Bengio. Science is a collaborative endeavour and I couldn't have accomplished half of what I did on my own.

Indeed, I often like to say that, during my PhD, I've contributed more to the field through the discussions I've had and the occasional helping hand I gave than through all of the papers I've written. The reverse is true; I've grown far more as a person and a researcher through my interactions with an amazing number of people, at McGill, at Mila, at conferences, and beyond, than through the research I've done. To borrow the words of Tolkien, *I don't know half of you half as well as I should like; and I like less than half of you half as well as you deserve.*

I'm deeply grateful for those moments with Carles Gelada, Charles Onu, Genevieve Fried, Harsh Satija, Ivana Kajic, Jean Harb, Joshua Holla, Joshua Romoff, Khimya Khetarpal, Michelle Lin, Neil Girdhar, Pierre-Luc Bacon, Sumana Basu, Veronica Chelu, and Vincent Antaki, for my pandemic *weekly bootstrapping* gang, Amy Zhang, Anthony Chen, Martin Klissarov, Maxime Wabartha, Melissa Mozifian, Nishanth Anand, and Wesley Chung, for my labmates, Alan Do-Omri, Andrei Lupu, Angus Leigh, Gheorghe Comanici, Jad Kabbara, Kian Kenyon-Dean, Martin Gerdzhev, Matthew Smith, Phil Bachmann, Philip Amortila, and Zafarali Ahmed, and for all the folks I've met at conferences.

Finally, I'm immensely thankful to my family. Annie, for instilling in me a sense of play and wonder. Yoshua, for planting the seeds of science in my mind. Patrick, for the creativity you awoke in me and all the music you've made me discover. Megan, for your loving support and your exceptional sense of humor. Merci!

Contents

Contributions to Original Knowledge **iii**

Contributions of Authors **v**

Contents **viii**

1 Introduction **1**

1.1 Context and Motivation 1

1.1.1 Cognition, Abstractions, and Scale 1

1.1.2 Reinforcement Learning 3

1.1.3 Automation and Humanity 4

1.2 Central Hypotheses and Outline 5

1.2.1 Challenges in Temporal Credit Assignment 6

1.2.2 Using Bootstrapping to Train Generative Models 7

1.2.3 Agents that Learn by Themselves 8

2 Technical Foundations **9**

2.1 Deep Learning 9

2.1.1 Machine Learning and Function Approximation 10

2.1.2 Deep Neural Networks 13

2.2 Agents and the Reinforcement Learning Framework 16

2.2.1 Markov Decision Processes 16

2.2.2 Value Approximation Methods and Temporal Difference 18

2.2.3 Policy Improvement Methods 18

2.3 Deep Reinforcement Learning 19

2.3.1 Foundational Works and Methods 20

2.3.2 Challenges of Large Scale Reinforcement Learning 22

3 Generalization and Temporal Difference **23**

3.1 Generalization in Supervised Deep Learning 24

3.1.1 Overparameterization and Memorization 24
3.1.2 Reframing the Question of Memorization 26
3.1.3 Recent Developments 28
3.2 Interactions Between Interference and Temporal Difference 30
3.2.1 Interference in Neural Networks 30
3.2.2 Temporal Difference changes Generalization Dynamics 31
3.2.3 The Instability of Deep Temporal Difference Learning 42
3.2.4 Derivations and Hyperparameters 50
3.3 Discussion 55

4 Optimization and Temporal Difference 57
4.1 Accelerated Methods, Bootstrapping and Staleness 57
4.1.1 On Supervised Learning Tools in Reinforcement Learning 57
4.1.2 Accelerated Methods and Staleness 59
4.1.3 Related Work 61
4.2 Correcting Staleness in Momentum 64
4.2.1 Identifying Bias 64
4.2.2 Correcting Bias 65
4.2.3 Derivation of the momentum correction 67
4.3 Empirical Results 71
4.3.1 Supervised Learning 71
4.3.2 Temporal Difference Learning 72
4.3.3 Hyperparameters and Architecture Choices 79
4.4 Discussion 82
4.4.1 Limitations 83
4.4.2 The Supremacy of Accelerated Adaptive Methods 84

5 Generative Models through Bootstrapping 85
5.1 Generating Diverse Rewards 85
5.1.1 A Primer on MCMC 87
5.1.2 Literature Review 88
5.2 Flow Networks 89
5.2.1 Setting Description 89
5.2.2 Flow Networks as Generative Models 92
5.2.3 Objective Functions for Flow Matching 95
5.3 Empirical Results 99
5.3.1 A (hyper-)grid domain 99

5.3.2 Generating small molecules . . . 103
5.3.3 Multi-Round Active Learning Experiments . . . 112
5.3.4 Revisiting the hyper-grid with RL tricks . . . 115
5.4 Discussion & Limitations . . . 118

6 Learning to Act and Understand Without Supervision **119**
6.1 Unsupervised Learning . . . 119
6.1.1 Disentanglement and Representation Learning . . . 120
6.1.2 Reinforcement Learning without Reinforcement . . . 121
6.2 Jointly Learning Features and Policies . . . 122
6.2.1 Quantifying Independence into Learning Objectives . . . 122
6.2.2 Expanding controllable factors to be continuous . . . 124
6.3 Empirical Results . . . 126
6.3.1 A simple gridworld . . . 126
6.3.2 Selectivity as an only objective . . . 127
6.3.3 Experiments on MazeBase . . . 128
6.4 Discussion . . . 132
6.4.1 Considering Instances and Object Diversity . . . 132
6.4.2 The (Poor) Dynamics of Joint Disentanglement . . . 133

7 Synthesis **134**
7.1 Summary of Contributions . . . 134
7.2 Limitations . . . 135
7.3 Future Work . . . 136

Bibliography **139**

1

Introduction

1.1 Context and Motivation

1.1.1 Cognition, Abstractions, and Scale

Humanity has long been in the business of creating more and more powerful tools, but perhaps its most distinctive feature shared with but a few species is its capability to create cognitive tools. Such tools simplify and accelerate the process of thinking, not only within individuals but also within groups and societies (Hutchins, 1995). Language allows information to flow faster, formal logic allows robust deductive reasoning, mathematics and physics provide us with powerful predictive models of our physical reality.

Computer Science now allows us to think about a different category of cognitive tools. By formalizing the very notion of information (Shannon, 1948) and what can be *computed* out of it and to what limits (Church, 1936; Turing et al., 1936), this field empowered humanity by offloading cognition itself to external tools, computers.

Yet, this transition is far from over. The first and most obvious cognitive tasks performed on computers were arithmetic and logic, done on larger scales and at faster speeds than humans are capable of. Today, computers are capable of a fantastic array of tasks: drawing virtual scenes with an uncanny resemblance to reality, analyzing text for mistakes, coordinating satellites orbiting our planet, simulating incredibly complex quantum mechanical systems, storing and communicating incredible amounts of information, and more feats than I can list here.

However impressive, these tasks rely on fairly simple tools of cognition (or rather, computation), that is, simple rules of logic and arithmetic that are arranged in complex ways and at massive scale. Tomorrow, the limit we will be breaking is that of abstract cognition.

Indeed, it was quickly realized that not all cognitive processes nicely fit the framework of logic and arithmetic (Minsky, 1961): intelligent beings are capable of inductive thought formed on the basis of fuzzy statistical beliefs, and these beliefs can be acquired, i.e. learned over time (Turing, 1950). That this process can be automated in a computer is,

in a nutshell, **Machine Learning**.

What of abstract cognition then? Are machines capable of dealing with complicated arrangements of such fuzzy references to the world? Impressive human-like conversations can be had today with computers trained with Machine Learning (Brown et al., 2020). Arguably, such a system possesses the capacity for abstract thoughts; it takes in language, an abstract encoding of ideas, and produces a language response that could have very well been produced by a human. In doing so, it must have handled complex abstract ideas and processed them in abstract ways to create novel sentences. Yet, two aspects indicate this is possibly not the case.

Indeed, such impressive and complex systems suffer from a scale problem and a generalization problem, both of which might be attributed to these systems' lack of some automated capability for abstract thought.

The scale problem is simply that progress for current systems seems to roughly follow a power-law (Kaplan et al., 2020): to increase the performance of the system by one unit one needs to multiply (think "double") the amount of computation it requires by a constant. This is not practical, at least for the constants that our current methods have, and is at odds with the little amount of data that biological intelligent systems seem to require to achieve the same tasks. This observation, coupled with the observation that we can train artificial systems to perform certain specific kinds of predictions better than humans (such as identifying dog breeds, Foret et al., 2021), but not other kinds of predictions (such as answering natural language questions about images, Zhang et al., 2021), suggest that a particular type of abstract reasoning does not scale well with current methods.

Generalization is the ability for a model to make correct predictions about novel situations. The generalization problem refers here to the type (and frequency) of errors that are made. When making predictions about things that have never been seen before, Machine Learning systems sometimes behave in inconsistent (Arjovsky, 2021) or catastrophically wrong ways (Szegedy et al., 2014). In other words, they sometimes fail to generalize from their past experience, for reasons still fundamentally beyond the field's grasp.

Interestingly, the difficulty in identifying such reasons seems to go hand in hand with the increased complexity of the models we use. The last two decades have seen one particular class of models gradually take over many parts of Machine Learning, but also of many other sciences. Artificial neural networks (McCulloch and Pitts, 1943; Rosenblatt, 1958; Fukushima, 1980; Rumelhart et al., 1988) have been found to be an extremely powerful and general way to model natural data, particularly in their large scale form, deep neural networks (LeCun et al., 2015; Goodfellow et al., 2016), which have been found to be capable of impressive generalization (Krizhevsky et al., 2012; Silver et al., 2017; Brown et al., 2020).

This latter aspect, generalization, concerns a large part of this thesis, as we will discuss

in §1.2.1. It seems likely that the abstract systems of the future will still rely on the systems that we currently use as their foundation (assuming these systems resemble the human mind; see e.g. Kahneman 2011 for literature inspiring current research). As such, understanding how and why Machine Learning systems generalize (or don't) seems of vital importance.

1.1.2 Reinforcement Learning

Another more specific kind of cognitive task is that of predicting behaviour: actions to take in a system over time. Such an aspect of Machine Learning also quickly became a concern in the field (Skinner, 1953; Minsky, 1961; Sutton, 1988) due to its inherently difficult challenges. A particular paradigm currently dominating the field is the framework of Reinforcement Learning (RL, Sutton and Barto, 2018), which sees behaviours of all complexities as nothing more than the maximization of a *reward signal* over time (Silver et al., 2021). Although simple, and almost all-encompassing, such a framework has led to some impressive feats of long term decision making by computers, perhaps most exemplified by Silver et al. (2017)'s Go playing system, beating humans when such a capability was considered to be still decades away.

Indeed, many problems can be usefully framed as Reinforcement Learning problems. The problem of playing a game for example, where a player receives positive reward upon winning and negative reward after loosing. Using these rewards, a player can *learn* to change (or *reinforce*) its strategy, i.e. its behaviour. Other problems include regulating glucose levels in diabetics over time (Yasini et al., 2009), controlling regulators of epilepsy (Panuccio et al., 2013), driving a car (Sallab et al., 2017), forming maximally informative trajectories for MRI scanners (Pineda et al., 2020), or personalizing content recommendations (Chen et al., 2019).

Because of its potential application to a wide array of important problems, studying Reinforcement Learning is an appealing path for the simultaneous advancement of many application fields.

In this thesis, we contribute to the comprehension of Reinforcement Learning, but more specifically, this thesis revolves around the concept of **bootstrapping** (Bellman, 1956; Sutton, 1988). While we define this concept formally in the next chapter, it's useful to think of bootstrapping as relying on predictions of the future to learn about the present.

Intuitively, this can work particularly well with generalization. Consider the behaviour of throwing a red ball; after the ball lands, I can observe its trajectory and learn to predict its parabolic curve. What happens as soon as I start swinging my arm to throw a blue ball? Its weight might be different, but even before it has left my hand, I can already

adjust my internal model by relying on my prediction of the blue ball's future, which I make by generalizing from the trajectory of the red ball. Through this mechanism, which compares what's going on in my hand as I throw the blue ball with my predictions of the future, I can adjust my mental model of reality much faster than if I had to wait for the blue ball to complete its trajectory.

This analogy transfers to longer time horizons as well. By using our past experience, combining it with our present experience, and comparing both, we can learn to answer such questions as, what actions should I take to brew my coffee? or, should I get a PhD?

Let's finish this discussion of Reinforcement Learning by discussing its inherent challenges alluded to above. Central to Reinforcement Learning is the concept of time. Even though some aspects of the future can be predicted to various degrees of accuracy by computers, the inherent randomness and exponential growth over time of the number of possible futures makes predictions about how to act fundamentally hard. The reverse is just as true: when an event occurs, the number of its possible causes can grow arbitrarily large depending on how far back in time we're willing to look. Fortunately, mechanisms such as bootstrapping allow us to reuse information we've previously acquired to learn, but this only helps us so much.

In practice, such difficulties make generalization, credit assignment, and exploration hard. As I will make more explicit later in this chapter, these are central questions to this thesis.

1.1.3 Automation and Humanity

It would seem amiss not to discuss the historical context in which this thesis is written. Let us stray for a moment from the main topic of this thesis.

Tools empower whoever uses them: for a fixed amount of calories expended, more work is done. Throughout history this has been both a curse and a blessing, depending on the inclinations of the people wielding the tools (Diamond, 1997). Knowledge allowed humans to build extraordinarily intricate structures and networks of now planetary scale, reduced suffering, and prolonged life (Harari, 2014). The introduction of Machine Learning into our set of everyday tools certainly also has been beneficial for humanity in a myriad of ways, and in many aspects will continue to be for the foreseeable future. For example, part of this thesis is concerned with finding more efficient tools to discover novel drugs.

At the same time, more efficient tools can mean more efficient manipulation and oppression. More efficient tools allow fewer individuals to affect (willingly or not) larger and larger numbers of people (Noble, 2018; Ribeiro et al., 2020). That this is done on purpose or not may not matter, as current automation tools propelled by Machine Learning

often simply reproduce the existing biases that exist in the societies in which they are built (Buolamwini and Gebru, 2018; Perez, 2019; Abid et al., 2021).

Indeed, the problems of *fairness* (Barocas et al., 2017) and *alignment* (Leike et al., 2018; Brundage et al., 2020) are active fields of research. Until we have a deep understanding of these facets of Machine Learning, it may be a dangerous game to work at improving the existing tools we have (which we know to be unfair by default). At the same time, we may not be able to achieve fairness and alignment until we improve the tools we have, or at least our understanding of them (the two often go hand-in-hand).

Even assuming that all is well and fair, cognitive and physical automation through Machine Learning poses a new fundamental problem to the organization of societies. Virtually all of humanity is now operating under local variants of the same economic system, one which directly relates labour and value (Samuelson and Nordhaus, 2007; Piketty, 2018). As we've just alluded to, automated tools increase the effective labour that one individual can do, directly or indirectly through ownership of the tool. This means that whoever creates or owns such tools will be immensely valuable, which might mean wielding immensely more influence over others (Gilens and Page, 2014; Birn, 2014). This is concerning, if one cares about equality and equity, considering the current economic system is ubiquitously perceived in Western societies as the only option, even by those opposed to it (Fisher, 2009; Pew Research Center, 2019). Although it has arguably served us well in the past, and certainly propelled most of humanity forward, it's not obvious that in the face of automation it will continue to do so. Considering the key role of automation in this equation, it is likely that computer science and computer scientists will play a key role in how this future unfolds.

Finally, it certainly isn't too early in history to begin to think about the *existential-risk* problem (§26.3 Russell and Norvig, 2002; Bostrom, 2002). Are we on the path to create tools so powerful that their misuse, voluntary or not, could significantly harm humanity? Furthermore, intelligence, sentience, and cognition appear to be intrinsically linked. It makes sense to seriously ask what will happen when we create an artificial sentient being with cognitive capacities well beyond our own (Bostrom, 2014). Again, it seems likely for computer science to play a crucial role in the future of homo sapiens.

1.2 Central Hypotheses and Outline

Let's now discuss the concrete problems that this thesis is trying to solve. After having laid down the technical foundations of this thesis in chapter 2, we will spend some time trying to understand the challenge of temporal credit assignment through bootstrapping

in chapters 3 and 4. Then, in chapter 5, we will discuss how it is possible to leverage the ideas behind bootstrapping to produce diverse reward-seeking generative models. Finally, in chapter 6, we will discuss the challenge of learning representations, useful models of the world, that would facilitate temporal credit assignment and solving Reinforcement Learning problems.

1.2.1 Challenges in Temporal Credit Assignment

Generalization is the ability to make useful predictions about situations one has never encountered before. Temporal credit assignment is the ability to attribute a cause, usually backwards in time, to a currently observed effect. In chapters 3 and 4 we will see that the two concepts are deeply tied to one another, and in what ways. In fact, both chapters revolve around the hypothesis that deep reinforcement learning (RL using deep neural networks) uses the wrong tools for temporal credit assignment.

Indeed, while the Reinforcement Learning framework is a powerful way of conceiving of problems, allowing us to apply RL methods to a wide variety of them, its generality comes at some cost. When dealing in problems that are related to time, some fundamental assumptions that are at the core of many Machine Learning are no longer applicable.

Take for example the concept of a *stationary* environment. In such a environment, the values that a Machine Learning model has to predict do not change over time. For an image classification system, a cat will forever be a cat. In a large number of Machine Learning methods, in particular those used *within* Reinforcement Learning methods (as subroutines or subcomponents), this stationarity assumption is central. In many Reinforcement Learning methods, this assumption is broken. For example, as an agent learns to play the game of Go, it may realize that certain strategies are more advantageous than others, and will start using them when playing new games. This change will affect the type of games that the agent will use to learn, and what it has to learn about will change. We say here that the agent is facing a non-stationary learning environment. Here the stationarity was self-induced, but there exist many more kind that may be external to an agent.

By using tools from Machine Learning that rely on these (now broken) assumptions, deep RL methods lack the robustness and guarantees that normally come with such tools. In an attempt to understand this phenomenon, in chapter 3, we first introduce a functional definition of generalization by contrasting it with that of pure memorization, and show that this provides valuable insights into the behaviours of deep neural networks leading to generalization. We then introduce the notion of *interference*, and show that it captures some structural problems that prevent deep reinforcement learning agents from accurately

learning and performing temporal credit assignment. In particular, we find that it is the learning method (rather than the problem being solved) which, through its reliance on bootstrapping, is at the root of behaviours we find to be radically different between RL and non-RL experiments.

Armed with this knowledge, we introduce a novel optimization method in Chapter 4 which is aware of some aspects of non-stationarity, and show that it helps with speeding up temporal credit assignment. More specifically, this method relies on correcting *staleness* in momentum-based optimization methods (Polyak, 1964). These methods use the accumulation of past information to learn faster, but as time passes such information becomes stale. This is doubly so in non-stationary learning situations, for which we provide a correction when the non-stationarity arises from bootstrapping. Since bootstrapping is about using predictions of the future to learn about the present, when such predictions change, the learning problem changes, inducing non-stationarity.

1.2.2 Using Bootstrapping to Train Generative Models

A strength of Reinforcement Learning methods is that they can be used solve problems that aren't necessarily traditionally viewed as reward-maximization problems. In chapter 5 we take this view at heart to solve a generative problem by taking inspiration from a core Reinforcement Learning mechanism: bootstrapping.

Let's take for example the problem of generating a molecule such that it has nice properties. Such a process has multiple time steps, since the molecule must be enumerated in some way (e.g. atom-by-atom), and has characteristics that can be turned into a scalar reward (how good the molecule is), and so can be solved with RL methods (Popova et al., 2019). Traditionally, this problem is solved with highly heuristic methods based on iterative sampling (McDonnell et al., 1995; Kawai et al., 2014). For example in evolutionary methods, an existing pool of candidates is iteratively randomly modified and filtered for its most promising candidates. In so-called Monte Carlo Markov chain (MCMC) methods, this modification is made to be adaptive in order to more quickly obtain a high-reward candidate set. The advantage of these methods is that they naturally produce *diverse* candidates, at some cost. By comparison, RL methods are typically designed to produce a unique maximally rewarding solution. How should we best solve the problem of generating diverse *and* high reward candidates?

In chapter 5, we show that applying either RL or iterative methods to this problem has some particular downsides. We also propose a novel algorithmic framework, named GFlowNet, which takes on the best parts of each of these methods. First, this framework is sequential like RL rather than iterative like MCMC or evolutionary methods, which comes

with several advantages (for example, sampling is amortized over the learning procedure). Second, we use bootstrapping combined with the notion of flow to obtain a learning objective that induces the diversity normally achieved by iterative methods, but which RL struggles to obtain due its fundamentally reward-maximizing nature. We then show that the proposed method achieves these goals in practice, both in restricted settings and in a large scale drug discovery problem.

1.2.3 Agents that Learn by Themselves

The third and final part of this thesis concerns learning representations (Bengio et al., 2013b), internal world models that interactive agents rely on to take decisions. A central hypothesis that is made here is that agents should want to learn to control their environment, and more specifically, to learn to identify which parts of their environment are independently controllable.

The problem of learning representations is central to the application of Reinforcement Learning to large scale problems. Typically, problems are defined by an environment and its reward function, and agents learn to behave entirely from this reward signal, which one can think of a supervision to the agent provided by humans. This can be a problem if this reward is very specific (i.e. the reward is 1 for a robot when it makes a good coffee, 0 at every other instant of its life), because it is very hard to discover. This can also be a problem if the reward is not yet known, but we'd still wish for an agent to learn about its environment by themselves, without explicit supervision, so that when we have a task for it, adapting to the task is fast.

As such, in the absence of reward, the most interesting thing to do for an agent seems to be to learn to explore the world and understand how it works. We postulate that one effective way for an agent to model the world in such circumstances is to learn a set of behaviours, and to tie each behaviour to an internal representation of what the behaviour is changing. Furthermore, we push the agent to learn these representations and behaviours such that the behaviours affect unrelated, or *independent*, aspects of the environment.

In chapter 6, we give a formal definition of what it means for representations to be *independently controllable*, propose several learning algorithms that aim to recover such representations, and show experimentally that they allow agents to recover the true controllable aspects of their environment. We show that such agents are capable of interesting abilities, such as planning about the future and how they can affect it directly.

2

Technical Foundations

In this chapter, we present some of the technical foundations on which the work presented in the rest of this thesis is based on. We define the shared technical language we will be using throughout, and defer to later chapters the exposition of relevant technical background unique to them. We finish this chapter by outlining the motivations for this work, in terms of these technical foundations.

Throughout the thesis we assume technical knowledge from the reader at the level of computer science undergraduate textbooks. In particular, we refer to linear algebra concepts (see Anton and Rorres, 2013), probability theory (see Wasserman, 2013), calculus (see Stewart, 2009), and set theory and algorithmics (see Brassard and Bratley, 1996).

2.1 Deep Learning

Something is said to be **learning** if it adapts, for some purpose, to the information available to it over some period of time. This definition is quite broad, but useful to keep in mind as a context in which *machine* learning arises. Evolution is a form of learning: DNA adapts to its environment for the purpose of self-replication over generations. Memory is a form of learning: animals retain maps of their environments to better survive over their lifetime.

Machine Learning (ML) refers to a broad category of algorithms, ran on digital machines, which achieve some purpose, some **objective**, by using the information or **data** they are provided to self-correct and improve on this objective over iterations of the algorithm.

This adaptation is commonly done by adjusting a set of **parameters** that inform the predictions made by such algorithms. This predictive mechanism is commonly referred to as a **model**. Just as scientific models help us describe some aspect of reality (for example, Newtonian physics and general relativity are models; weather forecasts are done with models) to some degree of fidelity, machine learning models help us predict something about *data* (for example, what the data contains, how it is generated, or what actions to take given data about the present).

Deep Learning (DL) refers to a type of **parameterization** and accompanying learning algorithms that are inspired by biological neural networks and biological cognitive mechanisms. **Artificial Neural Network** (ANN) models have emerged recently as a very useful structure, a type of model that seems capable of making more accurate predictions about some aspects of reality than previous types of models.

For an in-depth overview of machine learning and deep learning, we refer the reader to Bishop (2006a) and Goodfellow et al. (2016) respectively. In what follows, we present the aspects of machine learning and deep learning relevant to this thesis.

2.1.1 Machine Learning and Function Approximation

Machine Learning methods are typically concerned with approximating **functions**, mappings f from one space $\mathcal{X}$ to another $\mathcal{Y}$. This approximation is possible when we have **data**, **examples** of $(x, y) \in \mathcal{X} \times \mathcal{Y}$ pairs (note that y is not always given) that come from some "true" mapping f^* (for example human-provided labels). ML methods typically have two main parts. The first describes a **parameterization**, an algorithm that describes how to compute $f(x)$ for any x. The second describes a **learning** method, an algorithm that describes how to adjust $f(x)$ given a set of examples, with the goal that $f(x)$ approximates $f^*(x)$. This learning (or *training*) is done by changing internal **parameters** of f used to compute $f(x)$, which we will denote with θ throughout (typically $\theta \in \mathbb{R}^n$, where n need not be fixed during the lifetime of a model). This coupling of model and parameters is explicited as $f(x;\theta)$.

We say that a given $f(x;\theta)$ is a **model**, it makes (imperfect) predictions about some **input** $x \in \mathcal{X}$. $\mathcal{X}$ can take a variety of forms, but conceptually can be thought of as $\mathbb{R}^d$ in most cases (and in a way, $\mathbb{R}^d$ is a superset of any data representable on a computer, which is convenient for us). For a given parameterization, we call the set of all possible models, induced by the set of all possible parameters Θ (for example $\mathbb{R}^n$), the **hypothesis class** $\mathcal{H} = \{f(\cdot;\theta)|\theta \in \Theta\}$. A common problem in ML is choosing this hypothesis class, in particular choosing its size by fixing n when $\theta \in \mathbb{R}^n$. A naive view of this suggests two outcomes: when $|\mathcal{H}|$ is too small and $f^* \notin \mathcal{H}$, we say that $f \in \mathcal{H}$ is **underparameterized**; when $|\mathcal{H}|$ is too large and many hypotheses are "correct" (or appear correct) we say that f is **overparameterized**, that is conceptually when $|\{\theta \mid \forall x\ f(x;\theta) \approx f^*(x)\}| \gg 1$. What we exactly mean by *correct* can be a fairly involved question, but superficially we can imagine correctness as the ability for a model to make good predictions on a given set of data. Note that even if a parameterization is overparameterized, finding those optimal parameters can be hard (in fact it can be NP-hard, see Blum and Rivest (1992)).

Another choice that induces different hypothesis classes is the parameterization choice,

i.e. how each element of θ is used to compute $f(x;\theta)$. For example, we may choose a linear parameterization for f, writing $f(x;[a\ b]) = ax+b$, or a quadratic one, $f(x;[a\ b]) = ax^2+b$. In the next section (§2.1.2) we will discuss the characteristics of using deep neural network parameterizations.

As mentioned previously, learning is typically done to achieve some quantifiable **objective**, typically that of matching f^*. In ML, this objective value often has a particular interpretation, it is a **loss function** which we wish to minimize, usually through differentiation (more on this later). It is worth noting that the design of objectives itself is a non-trivial enterprise (see for example de Brébisson and Vincent (2015) or Jaderberg et al. (2017)), as objectives are often only useful proxies of some other desire. For example, in **classification** we wish to learn functions that map some input to a unique class. As such we could measure the accuracy of a given model: how often the model gives the correct answer when tested on a set of data points. We could also measure other things: the false positive rate, the average likelihood of the correct answer (i.e. how confident the model is about its prediction). In **regression**, instead of a unique label we want our models to predict some scalar quantity in $\mathcal{Y} \subseteq \mathbb{R}$. We could measure the mean squared error, but also the mean absolute error (or generally errors of the form $|f(x;\theta) - y|^p$). The right objective to choose often depends on the problem on which we wish to apply the model. These two problems, classification and regression, are examples of **supervised learning** methods (see Murphy, 2012, for a reference), where y is known.

To quantify how well a model is doing, this objective value is typically measured on two non-overlapping sets of data, the **training set** and the **test set**. The training set includes examples that are used by the learning algorithm to adjust the parameters, while the test set is only used to obtain an unbiased estimate of the expected objective value (over all possible inputs). The training set itself is usually subdivided into two subsets, one of which is the **validation set**. The latter is used not to adjust parameters but hyperparameters (such as n, the number of allowed parameters).

Generalization refers to the ability of a model to make correct predictions on inputs that it has never seen. The gap between the objective value on the training set and the test set is called the **generalization gap**, i.e. let D_{train} and D_{test} be these sets, and let $J(y,\hat{y})$ be the objective value for a data point (x,y), it is:

$$\begin{aligned} \text{Gap} &= \frac{1}{|D_{\text{test}}|} \sum_{(x,y)\in D_{\text{test}}} J(y, f(x;\theta)) - \frac{1}{|D_{\text{train}}|} \sum_{(x,y)\in D_{\text{train}}} J(y, f(x;\theta)) \\ &= J_{\text{test}} - J_{\text{train}} \end{aligned} \tag{2.1}$$

When the objective value on the training set is near-optimal, and the generalization gap is large (precisely what this means is problem-dependent), a model is said to be **overfitting**.

Conversely, if the gap is small but the training objective is far from its optimal value, a model is **underfitting**. We depict this visually in Figure 2.1.

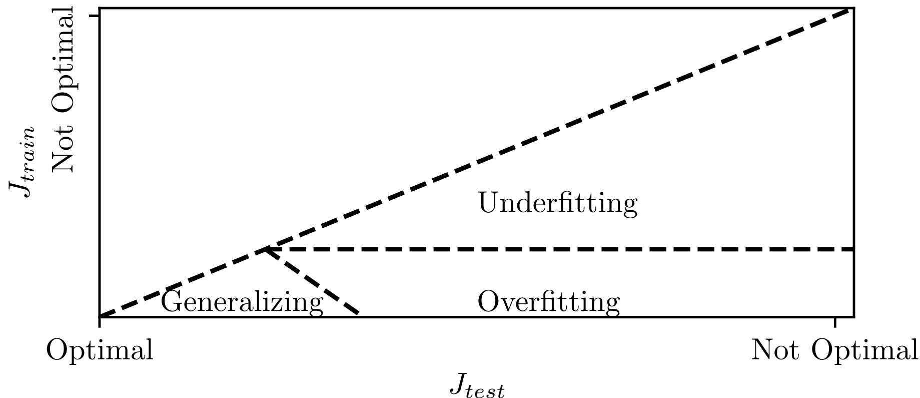


Figure 2.1: A cartoon view of overfitting and underfitting.

Generalization is often closely linked with the hypothesis class and the *amount* of data available: succinctly, the more data one has, the less likely it is for an overparameterized model to overfit.

Finally, the above discussion makes a very fundamental assumption about data. It assumes that the training set and test set are drawn from **independently identically distributed** (i.i.d.) random variables. In other words, it says that every example is drawn, independently from other examples, from some distribution $\mathcal{D}$. This is an important assumption to be mindful of for two reasons: first, it is often broken, and second, it is often a simplifying assumption. The so-called i.i.d. assumption is often broken, for example in online settings where data is collected on the fly, the distribution of data can change with time. This is often the case in Reinforcement Learning as we will see later (§2.3.2). The i.i.d. assumption is also often a simplifying assumption, data can often contain information about more aspects of the world than the task it is used for, other similar distributions we also care about. This is the subject of ongoing research (Arjovsky et al., 2019), and is also relevant in Reinforcement Learning.

Later in this thesis, we will discuss generalization in greater detail (§3.1), but note now that it is a central challenge in ML. Making predictions and developing theory about where models fail (especially complex models) and by how much they fail appears quite hard (Krishnapuram et al., 2005; Szegedy et al., 2014; Zhang et al., 2017; Dziugaite et al., 2020).

2.1.2 Deep Neural Networks

Artificial Neural Networks (ANNs) have become a standard parameterization for a large class of problems, often those concerned with so-called real-world data, such as sensory perception data–images (Krizhevsky et al., 2012), videos (Ranzato et al., 2014), sound (Oord et al., 2016), natural language (Devlin et al., 2018)).

Models are typically considered ANNs when they compose several simple parameterized transformations in a directed acyclic graph, most commonly linear projections and non-linear scalar transformations. Canonical ANNs (Rumelhart et al., 1988), also referred to as Multi-layer Perceptrons (MLPs), are expressed as a composition of functions of the form:

$$f(\mathbf{x};\theta) = (a_L \circ h_L \circ a_{L-1} \circ h_{L-1} \dots a_2 \circ h_2 \circ a_1 \circ h_1)(\mathbf{x}), \tag{2.2}$$

where a_l is a non-linear **activation** function applied element-wise to each component of a vector, and where a so-called **hidden layer** $h(\mathbf{v}; W, \mathbf{b})$ is a linear projection of the form $W\mathbf{v}+\mathbf{b}$. Common activations include the hyperbolic tangent, tanh, and the rectified linear unit, or ReLU: $a(x) = \max(0, x)$. For the model above, we write $\theta = \{W_1, \mathbf{b}_1, \cdots, W_L, \mathbf{b}_L\}$. Note that we shorthand $(h_l \circ \dots \circ a_1 \circ h_1)(\mathbf{x})$ as $h_l(\mathbf{x})$.

In the above example, we would say that f is a neural network with L *layers*. When L is large enough, we call these models **Deep Neural Networks** (DNNs). The distinction arises because, with **depth**, surprising (and useful) properties arise. DNNs are capable of expressing highly-complex functions (Montufar et al., 2014), of solving otherwise difficult problems (Krizhevsky et al., 2012; He et al., 2015), and possess intriguing generalization capabilities (Zhang et al., 2017) as well as scaling behaviours (Kaplan et al., 2020).

Again in the above example, note that the dimensions of the weight matrices can be arbitrary. For some function with $\mathbf{x} \in \mathbb{R}^{d_{in}}$ and $f(\mathbf{x}) \in \mathbb{R}^{d_{out}}$, as long as the dimensions of the product of these matrices is correct, i.e. that $W_L W_{L-1} ... W_2 W_1 \in \mathbb{R}^{d_{out} \times d_{in}}$, they can be of any shape. When W_1 has the shape $n_1 \times d_{in}$, we say that the **width** of the first layer is n_1, and so on for higher layers.

Just like width, depth, can be chosen somewhat arbitrarily, which induces a model design choice. Models with more width and depth can represent more complex functions, but require more storage and more computation power. They are also sometimes less convenient to train.

Notable variants of the above parameterization include convolutional neural networks (LeCun et al., 1998), which apply convolutions to the input to induce translation invariant predictions that are useful in visual and audio domains, transformer neural networks (Vaswani et al., 2017), which apply *attention* (Bahdanau et al., 2014) to selectively process their inputs, and recurrent neural networks (Rumelhart et al., 1988; Hochreiter and

Schmidhuber, 1997; Graves, 2013), which process their inputs sequentially, useful for domains such as natural language processing. While these variants are more efficient at dealing with various types of data, they all compose simple parameterized linear and non-linear transformations, and as such they all possess the properties mentioned previously. In that sense, they all belong to the class of DNNs. Since the work of this thesis does not rely on the particular properties of these parameterizations (rather, simply on the properties shared by all DNNs), we refer the reader to Chapters 9 and 10 of Goodfellow et al. (2016) as well as the works referred to above for more details.

Of particular interest in DNNs are the values that hidden layers take while computing $f(\mathbf{x})$. These values are referred to as the **hidden features**, or **representations**, or also **embeddings** (Bengio et al., 2013a), learned by the neural network. In particular, the value of the penultimate layer $h_{L-1}(\mathbf{x})$ is interesting. Naively, this level of representation can be seen as the most abstract representation of the input. This is typically the case due to the nature of the last layer h_L, for example, for an accurate classifier with $h_L(\mathbf{v}) = W\mathbf{v}+\mathbf{b}$, $h_L(\mathbf{x})_i$ being the unnormalized (log-)probability prediction for class i, the hidden representations h_{L-1} must be *linearly separable* (the points $h_{L_1}(\mathbf{x})$ of each class can be separated from every other class by a hyperplane). This is a potentially useful property, and allows for the reuse a models from one task to another (Bengio et al., 2007; Brown et al., 2020).

In the previous section we described *overparameterization* as there being a large number (or volume) of low-error solutions (where $f(x;\theta) \approx f^*(x)$) in the hypothesis class. For most problems, this is likely to be true for a large enough DNN. Indeed, Montufar et al. (2014) look at a way to quantify the complexity of functions that MLPs can express, and find an upper bound on this quantity. This bound is of the form $\Omega(c_1^{Ld}c_2)$, where $c_1 > 1, c_2 > 1$ depend on the width of the network, L is the number of layers and d the input dimension. The important aspect of this bound is the exponential dependency on L, the depth, which suggests that deep ANNs can express incredibly complex functions, and thus, are very likely to be in the **overparameterized** regime. More fundamentally, Cybenko (1989) and Hornik (1991) show that, given sufficient width, ANNs can express *any* function to an arbitrary degree of precision (but again, this doesn't mean that such ANNs are easy to find through learning and with limited information).

There is another important way in which DNNs are commonly thought of as overparameterized: the number of parameters commonly exceeds the number of training examples shown to a model. Common sense would suggest that in such a regime, the model is free to simply memorize what to predict on its training data with no regards to uncovering general patterns (i.e. it is not bound by Occam's razor (Blumer et al., 1987)) and thus will overfit and have a large generalization gap. Surprisingly, empirical results suggest the

opposite: larger, deeper models tend to systematically generalize better (Kaplan et al., 2020) if trained properly. In fact, there is now a wealth of literature trying to understand how and why DNNs generalize surprisingly well. We explore this literature in more detail in §3.1.

Let us conclude this section by discussing how DNNs are *trained*. We have so far avoided this subject because Machine Learning contains a plethora of training methods (and parameterizations). Such a variety also exists within the Deep Learning literature, but most of these methods are fortunately based around a simple idea: **gradient descent**. Recall how we formulated MLPs in (2.2), and note that as long as the activation functions a_l are differentiable functions, then $\nabla_\theta f(\mathbf{x};\theta)$, the so-called *gradient* of f, exists. By design, this is generally true of all DNNs. We noted earlier that there are many ways to design objectives; by choosing a differentiable objective function $J(\hat{y})$, such that $\partial J/\partial\hat{y}$ exists, we can compute the gradient $g(\mathbf{x}) = \nabla_\theta J(f(\mathbf{x};\theta))$ of the objective with respect to the parameters[1] θ. This gradient tells us how to modify θ in order to change (increase or decrease) J. By repeatedly following this gradient for a set (the training set) of $\mathbf{x}$s, we optimize for the objective J. When J represents a loss, something to minimize, then starting from some (usually random; Glorot et al., 2011) θ and iteratively following $-g(\mathbf{x})$ for randomly chosen training examples $\mathbf{x}$ with some speed α (the *learning rate*) is referred to as **stochastic gradient descent** (Bottou, 1998). Note that J may depend on more than the model prediction, as in supervised learning where target data y is given and we would write $J(f(\mathbf{x};\theta), y)$ or $J(\hat{y}, y)$.

From this simple idea a large number of optimization methods were developed, and we discuss these further in §4.1. One aspect to keep in mind is that these methods, gradient descent and its derivatives, are a priori only suited for (and designed for) strictly *convex* optimization problems (i.e. where the second derivative $\partial J/\partial\theta^2$ is always positive). While this property does not hold for DNNs, the *loss landscape* (how J varies with θ) of DNNs has interesting properties (Dauphin et al., 2014; Garipov et al., 2018; Fort and Ganguli, 2019) which make gradient-based methods work quite effectively, albeit sometimes requiring some hyperparameter tuning effort.

Deep Neural Networks have been at the center of a massive shift in how large scale algorithmic problems are solved. A significant part of this thesis explores how these models behave when used in conjunction with algorithms designed for interactive agents, and explores some problems that arise and some interesting solutions.

[1] Here, and throughout, we abuse notation slightly: when treating θ as a vector, we mean the concatenation of all scalar parameters contained in elements of set-θ. Similarly, here we'd consider $g(\mathbf{x})$ to be a vector of the same shape as vector-θ.

2.2 Agents and the Reinforcement Learning Framework

We have so far discussed the idea of *models* as fairly static predictive tools that transform information from one type to another, perhaps more useful type. When such models also predict *what to do*, we call those models **agents**. Such agents can be thought of as actors exercising their agency in an **environment**.

This definition is again quite broad, but perhaps aptly so, as many things can be accurately thought of as agents: bacteria, plants, animals, communities and countries, but also robots, content recommendation systems, and electricity grid management systems.

Reinforcement Learning (RL, see Sutton and Barto, 2018) refers to a broad category of algorithms concerned with analyzing and improving the behaviour of agents by learning from data. It is often considered a subset of Machine Learning, although in practice it is interconnected with a variety of other fields, ranging from neuroscience to economics.

What causes agents to act? The Reinforcement Learning framework makes a fundamental assumption about this, which is that agents act in order to maximize a **reward** signal over a period of time. For some agents, this is quite straightforward: a chess-playing program maximizes its odds of winning, i.e. the one-time reward it gets at the end of a game for winning it. For other agents, perhaps like bacteria, this is not so obvious. This induces two problems, finding the reward that an existing agent follows, or *designing* a reward so that an agent maximizing it behaves in desired ways. Parts of this thesis are concerned with the latter problem (see Chapter 6), but most of the thesis considers that the environments and their reward signals are simply given, already designed.

It may also not be desirable to simply *maximize* reward. Learning behaviours which vary the amount of reward without simply attempting the find the unique best thing to do may be useful when the reward is noisy, uncertain, or when diverse solutions are required (see Chapter 5).

For an in-depth overview of RL, we refer the reader to Sutton and Barto (2018). In what follows, we present a common formalism used to instantiate RL, as well as the basic learning methods used within that formalism that are relevant to this thesis.

2.2.1 Markov Decision Processes

A **Markov Decision Process** (MDP) (Bellman, 1957) is a standard formalism for the agent-environment paradigm. An MDP $\mathcal{M}$ is a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{T} \rangle$, with $\mathcal{S}$ the set of

states, $\mathcal{A}$ the set of actions, $\mathcal{R} : \mathcal{S} \times \mathcal{A} \to \Delta(\mathbb{R})$ the reward distribution[2] function, and $\mathcal{T} : \mathcal{S} \times \mathcal{A} \to \Delta(\mathcal{S})$ the transition distribution function.

This object $\mathcal{M}$ tells us what happens when an agent is in state $s \in \mathcal{S}$ and performs action $a \in \mathcal{A}$ by defining a distribution over possible scalar rewards that the agents can get from this action, and a distribution over possible states that the agent might end up after taking this action. Note that this distribution only depends on s and a, and not on states the agent might previously have been in, i.e. this process obeys the *Markovian* property (see Murphy, 2012, Ch. 17). A **policy** $\pi : \mathcal{S} \to \Delta(\mathcal{A})$ is a mapping from states to distributions over actions, and characterizes a behaviour for an agent. For convenience we write the probability of an action given a state as $\pi(a|s)$.

Let us now define a few key random variables. Let a **trajectory** $\tau(S_0, \pi) = (S_0, A_0, R_0, S_1, A_1, R_1, ..., S_H)$ be a sequence of length H of states, actions, and rewards, where $S_{t+1} \sim \mathcal{T}(S_t, A_t)$, $A_t \sim \pi(S_t)$, $R_t \sim \mathcal{R}(S_t, A_t)$. Note that H, the **horizon**, can be arbitrarily large and can itself be thought of as a random variable, as some formalisms of MDPs define a set of states to be *terminal* states which terminate trajectories. Other formalisms consider terminal states to simply be *absorbing*, they transition to themselves regardless of the action, and consider H to always be infinite (see Puterman, 1994, for an in-depth discussion of MDPs). We call the discounted sum of rewards

$$G(\tau) = \sum_{t=0}^{H-1} \gamma^t R_t, \tag{2.3}$$

the **return**, where $\gamma \in [0, 1)$ is a **discount factor**. Note that some formalisms of MDPs include γ a part of the tuple, or make it a function of the state, but it is not strictly necessary. We write $G(S)$ for the return of a trajectory starting in S.

Two key quantities emerge, the so-called **value** function v, and its action-dependent counterpart, the *action-value* function q, respectively defined as the *expected* return, when starting from some state, or starting from some state and taking some action:

$$v(s, \pi) = \mathbb{E}_\tau[G(\tau)|S_0 = s], \tag{2.4}$$

$$q(s, a, \pi) = \mathbb{E}_\tau[G(\tau)|S_0 = s, A_0 = a]. \tag{2.5}$$

An important observation is that these quantities can be expressed recursively. Here we express v and q for countable action and state spaces, but the same definitions extend to continuous spaces:

$$v(s, \pi) = \mathbb{E}_{R_t, A_t, S_{t+1}}[R_t + \gamma v(S_{t+1}, \pi)|S_t = s], \tag{2.6}$$

$$q(s, a, \pi) = \mathbb{E}_{R_t, S_{t+1}}[R_t + \gamma \sum_{a' \in \mathcal{A}} \pi(a'|S_{t+1}) q(S_{t+1}, a', \pi)|S_t = s, A_t = a]. \tag{2.7}$$

[2]We shorthand $\Delta(X)$ throughout to denote the type "a distribution over the set X".

In general we omit the dependency on π when unambiguous. It may also be convenient to subscript π as in v_π. The value functions can be useful when determining which states it is desirable to be in, or which action to pick given the current state.

2.2.2 Value Approximation Methods and Temporal Difference

Much like any other function, the value functions can be approximated by a model. For now, we assume that states have no information attached to them other than their identity. In §2.3 we explore methods that would use available information. When the number of states and actions is finite, we can simply store the predicted quantities in a look-up table; this is the tabular setting. In general we write V for approximations of v.

When we do not have access to $\mathcal{T}$ and only to *samples* of trajectories, we have no choice but to approximate the expectations in (2.6) and (2.7). There broadly exist two types of methods, based on two simple algorithms. The first type is **Monte Carlo** methods, which use entire trajectories to learn unbiased estimates. For example, in first-visit Monte Carlo, the value of a state s is computed to be the average of returns $G(s)$ found in trajectories for the first occurrence of s in those trajectories.

The second type, and most relevant to this thesis, is **Temporal Difference** (TD) methods (Sutton, 1988), which only use parts of trajectories. Its simplest method, 1-step TD, or TD(0), iteratively uses each *transition* (S_t, A_t, R_t, S_{t+1}) to update the stored value $V(S_t)$ with some learning rate α with the following update:

$$V(S_t) \leftarrow V(S_t) + \alpha[R_t + \gamma V(S_{t+1}) - V(S_t)]. \tag{2.8}$$

Under this method, the value estimate of a state $V(s_t)$ uses the value estimate of the *next* state $V(s_{t+1})$ to improve itself. This is called **bootstrapping**. TD methods are interesting because, in a sense, they allow for an efficient use of experience. Much of this thesis is dedicated to understanding how TD methods interact with deep neural networks, as this combination has yielded valuable results, but also interesting failures.

2.2.3 Policy Improvement Methods

So far we have discussed interesting quantities about MDPs, assuming some policy π was given, or at least that we had data generated by an agent following some fixed policy, but it is possible to learn and approximate policies as well.

A large part of RL is composed of methods to find reward-maximizing policies. These methods are typically policy *improvement* methods; they take in a policy and output a better policy. Generally, the iteration of such improvements lead to an *optimal policy.* In

MDPs there exists a unique optimal value v^*, to which corresponds a set (possibly of size 1) of optimal policies π^* such that $v^*(s, \pi^*) = \max_\pi v(s, \pi)\ \forall s$.

There are two notable classes of policy improvement methods. The first is based on **greedy** policy improvement, which computes the value of the current policy π_k, q_{π_k}, and outputs $\pi_{k+1}(a|s) \leftarrow \arg\max_u q(s,u)$ or equivalently $\arg\max_u \sum_{S_{t+1}} P(S_{t+1}|S_t = s, A_t = u)v(S_{t+1})$. In other words, it creates a policy that greedily acts according to its current estimates of how good actions are.

The second class is that of **policy gradient** methods (Sutton et al., 1999a), which, as the name suggests, improve *parameterized* policies by computing the gradient of an objective with respect to the policy parameters θ. Unlike for the above greedy deterministic policies, a distribution over actions is stored[3] instead of the best action. This permits the gradient of the *return* w.r.t. θ to be expressed in terms of adjustments to the probability of each action. The canonical policy gradient (Williams, 1992a) formulation is of the form:

$$\mathbb{E}_\tau[\nabla_\theta G(S_0)] = \mathbb{E}_\tau[G(S_0) \sum_{t=0}^{H-1} \nabla_\theta \log \pi_\theta(A_t|S_t)]. \tag{2.9}$$

Note that this expectation is over *all possible trajectories* τ, and is typically intractable, because the number of trajectories starting from some state S_0 with horizon H is typically in $O(|\mathcal{A}|^H)$, i.e. it is exponentially large. Thus, the expectation has to be approximated from a comparatively much smaller number of samples. This leads policy gradient methods to suffer from variance problems, whereby Monte-Carlo estimates of the gradient are typically so high variance that a large number of samples would be required to obtain coherent estimates (Ilyas et al., 2018), although in practice this does not always prevent them from performing well, as we will illustrate in the next section.

2.3 Deep Reinforcement Learning

In Section 2.1 we have presented Deep Neural Networks; they are a powerful set of modelling tools to approximate functions. In Section 2.2 we have presented the Reinforcement Learning framework, which is a general way to conceptualize interactive agents, as well as some classes of common learning methods to evaluate and improve the behaviour of agents.

Deep Reinforcement Learning (DRL or Deep RL) refers to the category of methods that use DNNs in order to approximate functions necessary to model interactive agents.

[3]There also exists so-called deterministic policy gradient methods (Silver et al., 2014), but they are beyond the scope of this thesis.

This combination opens up a myriad of possibilities in terms of what types of problems can be solved, and how fast.

In section 2.2 we have discussed *tabular* methods, that assume all we know about a state is its identity. This makes generalization impossible: unless we have information about every state and every action, tabular models cannot make predictions about novel situations. The MDP framework makes no specific assumptions about the nature of the state space $\mathcal{S}$, but becomes much more interesting when we allow $\mathcal{S}$ to have some *structure*, for example, if $\mathcal{S} \subset \mathbb{R}^d$ represents a set of continuous variables representing some *information* about the state. Such a setting is interesting because it makes *generalization* possible, since it allows the use of function approximation using models, and so enables models to make predictions about novel situations.

In the following sections we present some notable deep RL methods, the type of problems that they help solve and some of the challenges that arise.

2.3.1 Foundational Works and Methods

The first notable uses of an ANN within an RL algorithm date back to Barto et al. (1983), (Schmidhuber, 1991), and Tesauro (1995). In particular, Tesauro's work implements a particular form of Temporal Difference learning, TD(λ) (Sutton, 1988; Munos et al., 2016), using a neural network to approximate the value function v, to play the game of Backgammon. We previously discussed approximating v with a tabular V. More generally, we can use any parameterized model that takes as input descriptions of states to approximate v, and write $V_\theta(S)$ for a model with parameters θ.

Recall the recursive definition of the value function in (2.6):

$$v(s) = \mathbb{E}[G_t|S_t = s] = \mathbb{E}_{R_t,A_t,S_{t+1}}[R_t + \gamma v(S_{t+1})|S_t = s].$$

When approximating v with V_θ, our goal is to perform a regression to G, or equivalently using a weighted averaging of future targets called a λ-return:

$$G^\lambda(S_t) = (1-\lambda)\sum\nolimits_{n=1}^{\infty} \lambda^{n-1} G^n(S_t) \tag{2.10}$$

$$G^n(S_t) = \gamma^n V_\theta(S_{t+n}) + \sum\nolimits_{j=0}^{n-1} \gamma^j R(S_{t+j})$$

$$\mathcal{L}_{TD(\lambda)}(S_t) = (V_\theta(S_t) - G^\lambda(S_t))^2. \tag{2.11}$$

Note that the return depends implicitly on the trajectory and the actions followed from S_t when following some policy π. When $\lambda = 0$, the loss is simply $(V_\theta(S_t) - (R_t + \gamma V_\theta(S_{t+1})))^2$, leading to the TD(0) algorithm (Sutton, 1988).

While TD(λ) is useful to *evaluate* a policy π, if we are interested in finding the best policy π^*, two families of methods that are compatible with ANNs emerge: Q-Learning methods (Mnih et al., 2013) and deep policy gradient methods (Mnih et al., 2016).

Recall now the definition of the action-value function in (2.7). The action-value function of the *greedy policy*:

$$\pi(a|s) = \mathbf{1}[\text{argmax}_u Q(s,u) = a], \tag{2.12}$$

can be written conveniently as:

$$q(s,a,\pi) = \mathbb{E}_{R_t,S_{t+1}}[R_t + \gamma \max_{a'} q(S_{t+1}, a')]. \tag{2.13}$$

This yields the well known Q-Learning algorithm (Watkins and Dayan, 1992) and its modern deep learning incarnation (Mnih et al., 2013), DQN. In such a method, we once again perform a regression:

$$\mathcal{L}_{QL}(S_t, A_t) = (Q_\theta(S_t, A_t) - (R_t + \gamma \max_{a'} Q_\theta(S_{t+1}, a')))^2, \tag{2.14}$$

where once again this implicitly depends on S_{t+1} via sampling of the policy.

Recall the **policy gradient** (PG) formulation in (2.9). This gradient computation can be adapted to DNNs in a few straightforward ways. In particular, to reduce the reliance on computing gradients over entire trajectories (as required by $G(S_0)$), Mnih et al. (2016) adapt the so-called actor-critic formulation to the deep case by relying on a learned value function to perform the policy gradient update:

$$\mathbb{E}_{S_t}[\nabla_\theta G(S_t)] \approx \mathbb{E}_{S_t}[(\hat{G}_t^{(k)} - V_\theta(S_t))\nabla_\theta \log \pi_\theta(A_t|S_t)]. \tag{2.15}$$

with $\hat{G}_t^{(k)}$ a *truncated* k-step return, $\hat{G}_t^{(k)} = \gamma^k V_\theta(S_{t+k}) + \sum_{i=0}^{k-1} \gamma^i R_i$.

Another notable PG method, used in multiple chapters, is Proximal Policy Optimization (PPO, Schulman et al., 2017). PPO works by maintaining two sets of parameters θ and θ_{old}, used to parameterize a policy π. θ is continually updated, while θ_{old} is only updated at regular intervals, and is used to collect data (with $\pi_{\theta_{old}}$). PPO learns a stable policy by forcing it to stay close to a policy it "already knows" works, or in other words, it forces π_θ to stay close to $\pi_{\theta_{old}}$. This is achieved by using the following objective:

$$\mathcal{L} = \mathbb{E}_{S_t}\left[(G_t - V_t)\frac{\pi_\theta(A_t|S_t)}{\pi_{\theta_{old}}(A_t|S_t)}\right], \tag{2.16}$$

and by preventing this objective of updating π_θ to be to far from $\pi_{\theta_{old}}$. One straightforward way of doing so is by simply clipping the policy ratios. The objective proposed by Schulman et al. (2017) is thus:

$$\begin{aligned} r &= \frac{\pi_\theta(A_t|S_t)}{\pi_{\theta_{old}}(A_t|S_t)} \\ \mathcal{L}^{CLIP} &= \mathbb{E}_{S_t}\left[\min\left(r(G_t - V_t), \text{clip}(r, 1-\epsilon, 1+\epsilon)(G_t - V_t)\right)\right]. \end{aligned} \tag{2.17}$$

PPO is considered a state-of-the-art method for many problems, and commonly used as a baseline.

Note that we haven't addressed several details necessary to implement these methods, e.g. how Q and V should be parameterized, where the data used for the updates above should come from, what optimization method should be used. These details are important, but vary from problem to problem, and are the principal content of a large number of recent publications. Later in this thesis, we explore some of the problems of these methods, in particular those tied to bootstrapping, at which point we will explicitly describe these details.

2.3.2 Challenges of Large Scale Reinforcement Learning

Although the algorithms described so far are seemingly capable of solving increasingly complicated problems using deep neural networks, two principal challenges have emerged from the literature.

First is one of *generalization*. Even though DNNs have remarkable generalization capabilities, deep RL agents have been found to be very brittle (Farebrother et al., 2018), incapable of adapting to minor changes in their environment. One of the reasons for this brittleness appears to be that the tools (the learning methods, model parameterizations, data processing) we use in deep RL are naively imported from supervised learning, where for example data is ideally distributed–this is not the case in RL, since as an agent learns its data distribution changes dramatically, hence not independently distributed.

The subject of Chapter 3 is to identify where these generalization problems take root, while Chapter 6 concerns methods that aim to learn general representations of environments.

The second challenge is one of *sample complexity*. It takes incredible amounts of data to solve problems using the most efficient deep RL methods known so far. For example, AlphaGo Zero (Silver et al., 2017), while on-par with or capable of beating every human expert, required 29 million virtual games against itself – around 40 days of training on specialized hardware, but the equivalent of hundreds of human lifetimes (about 200 40-year careers of 8 hours of daily play). This phenomenon is not unique to Go, on simple arcade environments such as Atari games (Bellemare et al., 2013), agents are routinely trained for hundreds of millions of frames – almost a thousand hours of real-time play. In Chapter 4 an optimization method is presented which attempts to curb this complexity.

3

Generalization and Temporal Difference

This chapter synthesizes two of my contributions, my partial contribution to the work of Arpit et al. (2017), as well as my own work on interference and generalization (Bengio et al., 2020a).

Defining **generalization** is no easy task. While some simple formulations of generalization can help us compare different models with some confidence, they do not provide the breadth of guarantees that algorithm designers look for in their methods, nor do they help us explain *why* certain models behave one way or another.

Here we focus on this latter aspect, the behaviour of models. We first ask what are interesting behaviours to look for, and then propose some insights on how different methods affect these behaviours, and thus generalization.

Through empirical analysis, we find interesting refinements on our tool set to understand neural networks. One particular aspect of deep neural networks is their capability for memorization; while neural networks do heavily rely on certain training examples (in a sense, *memorizing* them), it seems this behaviour is not incompatible with generalization. Another aspect of deep neural networks is the strong interconnectedness between parameters, where any given parameter is responsible for a large number of inputs, and conversely, a single input/output is affected by a large number of parameters. This behaviour, referred to as *interference*, affects how neural networks learn and generalize. Interestingly, when this learning is done through temporal difference, the relationship between interference and generalization *changes*, in some cases reversing, which has intriguing consequences for learning.

In the following sections we expand on these results, first providing some context for the investigation of generalization, and then discuss how these results might affect what research directions should be considered by our field.

3.1 Generalization in Supervised Deep Learning

3.1.1 Overparameterization and Memorization

Let's first recall how we defined generalization in §2.1.1, where we said it is the ability of a model to make correct predictions on inputs that it has never seen. Concretely, in supervised learning this is often measured through the *generalization gap*, the difference in performance on the training set, examples that the model has *seen*, and the test set, which the model has never seen:

$$\begin{aligned} \text{Gap} &= \frac{1}{|D_{\text{test}}|} \sum_{(x,y)\in D_{\text{test}}} J(y, f(x;\theta)) - \frac{1}{|D_{\text{train}}|} \sum_{(x,y)\in D_{\text{train}}} J(y, f(x;\theta)), \\ &= J_{\text{test}} - J_{\text{train}}. \end{aligned} \tag{3.1}$$

In learning theory, several quantities *predict* this generalization gap by relating the complexity of the input space with the complexity of the model (Vapnik and Chervonenkis, 1971). Superficially, we can understand this relationship as predicting what model complexity is necessary (how many parameters one can use) given the dimensionality of the inputs on the number of training examples available. Such a prediction suggests that if a model has too many parameters, it will have a large generalization gap.

As DNNs emerged as a parameterization of choice to solve increasingly complicated tasks, this posed a problem. Why do models with orders of magnitude more parameters than both input dimensions and number of training examples generalize so well? This is not what traditional learning theory suggests.

One of the traditional views of overparameterization is that it induces a hypothesis class where many models fit the training data *exactly*, and that when training an overparameterized model (i.e. performing a search within this hypothesis class) to convergence, one essentially finds one of these models *at random*. This suggests that in-between training data points, the interpolation could be anything – most likely nonsense.

This view is often depicted visually with a polynomial regression. Consider the regressions in Figure 3.1; a polynomial of degree $p = 3$ ($f(x;\theta) = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3$) is sufficient to fit these 4 data points and obtain a "good looking" curve. We could use higher degree polynomials, in fact if we're smart about it we can even use $p = 100$ and get reasonable results. But using so many parameters for such a simple problem is a dangerous game, as we can see in Figure 3.1-Right, naively finding $p = 100$ solutions that fit the data can yield models that make terrible predictions in-between data points.

This is a traditional view of the *memorization* hypothesis, namely that for overparameterized models that fit the data perfectly, we should not expect good generalization because

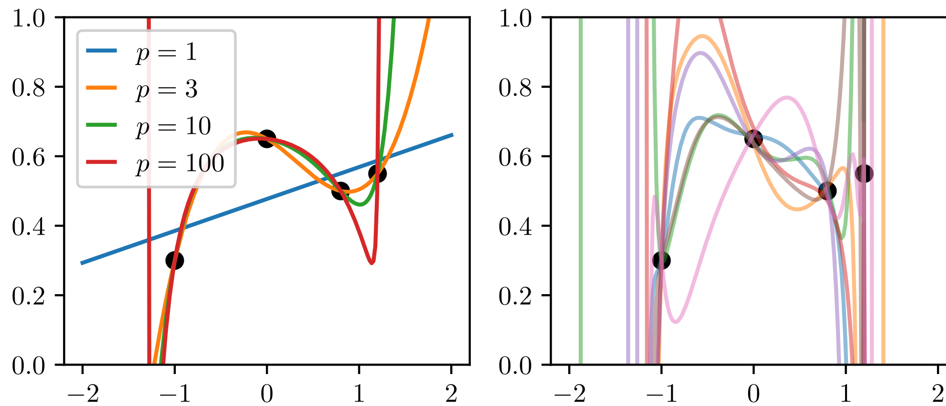


Figure 3.1: Left: a standard polynomial regression with an increasing number of degrees, p. Right: Randomly sampled polynomials of degree 100 that also perfectly fit the 4 data points.

the model has simply memorized its training examples. A corollary of this hypothesis is that a model that has a low generalization gap *does not memorize*, and instead relies and generic features, general aspects of the data which are useful for prediction beyond the training set. Thus, for a long time, it was assumed that DNNs find such general features and avoid memorization, explaining their great performance. As it turns out, this is not the right way to think about DNNs.

In their seminal paper, Zhang et al. (2017) show that DNNs can be trained to make arbitrary (or random) zero-error predictions on random data, without significant additional computational effort.

Let us focus on one particular experiment of Zhang et al. (2017), where deep convolutional networks are trained on natural images to predict shuffled labels. In this setup, the DNNs are *forced to memorize* the image-to-label mapping, since it is arbitrary. Surprisingly, training such a model to near-zero loss takes only about $2\times$ the number of learning steps than training on the real labels.

This result was surprising when it came out, because it suggested the following: optimizing a DNN to memorize fake data isn't much harder than optimizing a DNN to "generalize" to real data, i.e. reach a low test-data error, and so DNNs trained on real data are probably in a "memorization" regime as well. If this suggestion were true, then if would leave us with a puzzle: understanding why overparameterized models that do in fact memorize still have a low generalization gap.

3.1.2 Reframing the Question of Memorization

In Arpit et al. (2017), we attempt to better understand this puzzle. Let us focus on two central findings of the paper, (1) that DNNs learn simple patterns first, and (2) that DNNs *behave* differently when trained on *fake* data. Here by *fake* data we mean data that has been corrupted to remove its correlations, for example by assinging random labels to natural images as in the experiment of Zhang et al. (2017) we just highlighted.

These findings allowed us to reframe this puzzle and claim something important: memorization and generalization are not at odds. They show that DNNs have two underlying phases during training: first they learn simple patterns, and then they adjust their parameters to find the patterns specific to individual examples to reach near-zero error. This suggests that, even though DNNs do rely on memorization, their predictions on unseen examples is likely to rely on general features. This enables them to have a low generalization gap.

The finding that DNNs behave differently when trained on fake data is also useful to support this last hypothesis. Even though DNNs are *capable* of memorizing anything as shown by Zhang et al. (2017), if they are trained on data that has structure, they appear to be in a functionally different regime. As an analogy with Figure 3.1, when the data has structure, DNNs are more likely to find solutions which are smooth and general (as in the solutions on the left of the Figure) rather than more "arbitrary" solutions (as the ones on the right).

I will now highlight my particular contribution to this paper, which came to life as a fusion of multiple follow up experiments to the work of Zhang et al. (2017) from a total of 11 researchers.

Measuring the influence of individual training examples One way to show that DNNs are in a different regime when trained on fake data is to try to measure if individual data points are memorized. In this experiment, we propose a proxy for memorization.

Since we cannot measure quantitatively how much each training sample $\mathbf{x}$ is memorized, we instead measure the *effect* of each sample on the average loss. To do so, we measure $g^t_{\mathbf{x}_j}$ the norm of the gradient of the loss, $\mathcal{L}_t$, with respect to a given example $\mathbf{x}_j$, $j < t$ after t SGD updates. Let $\mathcal{L}_t = \mathcal{L}(\mathbf{x}_t; \theta_t)$ be the loss $\mathcal{L}$ (for example here a classification loss) computed on $\mathbf{x}_t$ after t SGD updates to θ, these updates have the form:

$$\theta_{t+1} = \theta_t - \alpha \nabla_{\theta_t} \mathcal{L}(\mathbf{x}_t; \theta_t), \tag{3.2}$$

and so this measure is given by

$$g^t_{\mathbf{x}_j} = \|\partial \mathcal{L}_t / \partial \mathbf{x}_j\|_1 \ . \tag{3.3}$$

Here, we can compute this quantity because $\mathcal{L}_t$ is a differentiable function of θ_t, itself a differentiable function of θ_{t-1}, and so on until θ_{j+1} where $\mathbf{x}_j$ is the example used to perform the update at $t = j$. In other words, the parameter update from training on $\mathbf{x}_j$ influences all future $\mathcal{L}_{t>j}$ indirectly through θ by changing the subsequent updates on different training examples. In what follows we drop the j subscript for clarity. In practice, we compute $g^t_{\mathbf{x}}$ by unrolling t SGD steps and applying backpropagation over the unrolled computation graph, as done by Maclaurin et al. (2015). Unlike Maclaurin et al. (2015), we only use this procedure to compute $g^t_{\mathbf{x}}$, and do not modify the training procedure in any way.

We denote the average $g^t_{\mathbf{x}}$ on the training set after T steps as $\bar{g}_{\mathbf{x}}$, and refer to it as *loss-sensitivity*.

We run experiments using a fully-connected network with 2 layers of 16 units on 1000 downscaled 14×14 MNIST digits using 10^5 SGD steps, and find that for real data, only a subset of the training set has high $\bar{g}_{\mathbf{x}}$, while for random data, $\bar{g}_{\mathbf{x}}$ is high for virtually all examples. We also find a different behaviour when *each example* is given a unique class; in this scenario, the network has to learn to identify each example uniquely, yet still behaves differently when given real data than when given random data as input.

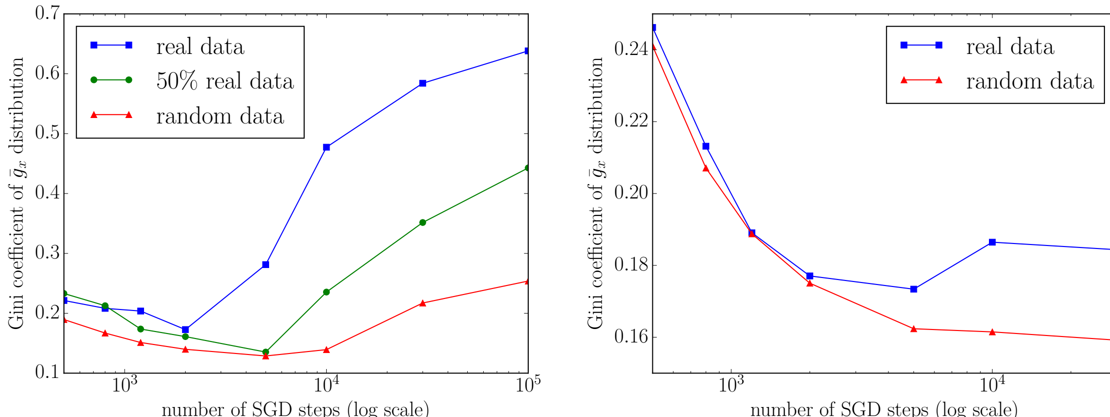


Figure 3.2: Plots of the Gini coefficient of $\bar{g}_{\mathbf{x}}$ as training progresses, for a 1000-example real dataset (14x14 MNIST) versus random data (uniform noise). On the left, y is the normal class label; on the right, there are as many classes as examples, the network has to learn to map each example to a unique class.

We visualize in Figure 3.2 the spread of $\bar{g}_{\mathbf{x}}$ as training progresses by computing the Gini coefficient over $\mathbf{x}$'s. The Gini coefficient (Gini, 1912) is a measure of the inequality among values of a frequency distribution; a coefficient of 0 means exact equality (i.e., all values are the same), while a coefficient of 1 means maximal inequality among values:

$$\text{Gini}(p \in \Delta(X)) = \frac{\sum_i \sum_j |p(x_i) - p(x_j)|}{2|X| \sum_i p(x_i)}. \tag{3.4}$$

We observe that, when trained on real data, the Gini coefficient is high (the network has a high $\bar{g}_{\mathbf{x}}$ for a few $\mathbf{x}$s, and a low for other $\mathbf{x}$s) suggesting the network is sensitive to a select few examples, while on random data the network is sensitive to most examples–the Gini coefficient is low.

The difference between the random data scenario, where we know the neural network needs to do memorization, and the real data scenario, where we're trying to understand what happens, leads us to believe that this measure is indeed sensitive to memorization. These results suggest that when being trained on real data, the neural network is in a different learning regime. Nonetheless, this regime implies a larger amount of reliance on a few select training examples, which implies that some form of *memorization* is happening.

The way we interpret these results is by associating this low Gini coefficient with memorization: if all examples are equally important to the model, then in some sense, it is likely that the model has memorized those examples into its parameters. On the other hand, for a high Gini coefficient, it is likely that after having seen a few critical examples (because, remember, the optimization process of SGD is sequential), those examples are enough to have embedded *general* patterns into the parameters of the model; thus a high Gini coefficient can be interpreted as being able to generalize from a few examples. The semantics of whether reliance on these few examples should also be called *memorization* is debatable. At the very least it is memorization of a different kind.

In addition to the different behaviours for real and random data described above, we also consider a class specific loss-sensitivity:

$$\bar{g}_{i,j} = \mathbb{E}_{(\mathbf{x},y)|y=j} \frac{1}{T} \sum_{t=1}^{T} |\partial \mathcal{L}_t(y=i)/\partial \mathbf{x}| \tag{3.5}$$

where $\mathcal{L}_t(y=i)$ is the error on the prediction of probability of belonging to class i. We observe that the loss-sensitivity w.r.t. class i for training examples of class j is higher when $i=j$, but more spread out for real data (see Figure 3.3). An interpretation is that for real data there are more interesting cross-category patterns that can be learned than for random data.

3.1.3 Recent Developments

Understanding that memorization and generalization are not at odds has led to a number of developments in our comprehension of how and why DNNs perform the way they do. Let's briefly highlight some recent developments that stem from this research.

In the work of Belkin et al. (2019), the authors find a new strange phenomenon that again runs counter to the traditional insights of learning theory. When increasing the width of commonly used DNNs which achieve low test error, it is observed that the test

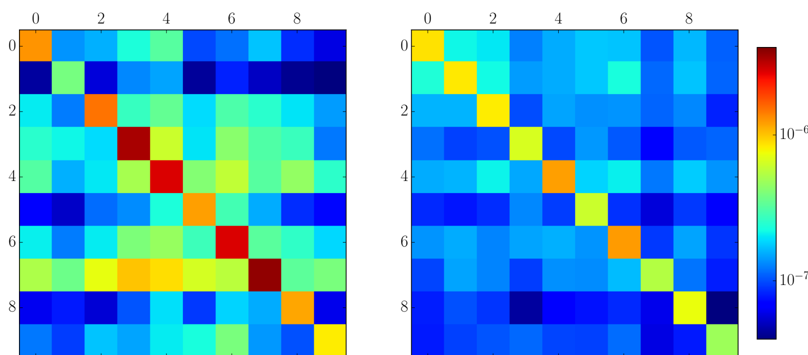


Figure 3.3: Plots of per-class sensitivity (log scale), a cell i, j represents the average loss-sensitivity of examples of class i w.r.t. training examples of class j. Left is real data, right is random data.

error first increases (as predicted by learning theory) but then decreases monotonically. Nakkiran et al. (2019) even find that this happens when simply training for very long. In either case, the training error is close to zero, again suggesting that these DNNs have memorized their training set while still making accurate generalizing predictions.

This so-called *double descent* phenomenon shows that something interesting happens with overparameterization. While increasing the number of parameters increases the size of the hypothesis class, it also changes the likelihood of encountering certain parameterizations after training. The hypothesis proposed by Belkin et al. (2019) is simply that having more parameters induces a smoother prior in randomly initialized DNNs (as the neurons' predictions are averaged with each other and there are more neurons, by the law of large numbers their average tends to a smooth distribution), and that due to the large number of parameters, SGD doesn't require a large change in individual parameters. Since the norm of this change is small, then the smoothness is conserved, which is beneficial for generalization in high-dimensions.

Relatedly, Frankle and Carbin (2018) find the existence of so-called *lottery tickets* in DNNs: subsets of parameters in large models which, at initialization, would have been enough to train a good model without having to use the rest of the parameters. The existence of these lottery tickets suggests that increasing the number of parameters is akin to drawing more often from the lottery, increasing the odds of having a winning ticket, and leaving more parameters free to memorize idiosyncrasies of harder examples. This again partly explains the success of overparameterization and memorization.

Studying these phenomena is important, not just for the emergence of novel better performing DNNs, but also to understand how memorization plays a role in how deployed DNNs affect the real world. Hooker et al. (2020) highlight how crucial this is by showing

that pruned DNNs (models whose "less important" parameters are removed for efficiency in deployment) end up performing poorly on a heavy-tail of outliers. This can have serious impacts on fairness for these models.

3.2 Interactions Between Interference and Temporal Difference

So far we have discussed generalization in supervised learning and deep learning, how it is measured, and how preconceptions and intuitions about it evolved over the last few years.

We have also previously mentioned that deep Reinforcement Learning has been found to be brittle, not robust to changes. Here, we investigate this question by focusing on a basic mechanism of RL, Temporal Difference learning, and how it affects generalization and memorization. We again attempt to change the lens through which people understand generalization and learning, in deep RL.

In the next sections, we closely follow the original material of my contribution on this topic (Bengio et al., 2020a).

3.2.1 Interference in Neural Networks

We first start by introducing a concept central to this investigation, **interference**. The interference between two gradient-based processes, each with objectives J_1, J_2, which share parameters θ, is often characterized in the first order by the inner product of their gradients:

$$\rho_{1,2} = \nabla_\theta J_1^T \nabla_\theta J_2, \tag{3.6}$$

and can be seen as the *influence*, constructive ($\rho > 0$) or destructive ($\rho < 0$), of applying a gradient update using $\nabla_\theta J_1$ on the value of J_2.

It's important here to understand the nature of the sign of interference. Rooted in physics (waves can constructively or destructively interfere with each other), this interpretation brings attention to the dual nature of learning: learning about something can help reinforce other similar concepts, but can also adversely affect the parameters used to remember other concepts, i.e. it can induce forgetting.

Interference as defined in (3.6) arises in a variety of ways: it is the interference between tasks in multi-task and continual learning (Lopez-Paz and Ranzato, 2017; Schaul et al., 2019), it forms the Neural Tangent Kernel (Jacot et al., 2018), it is the Taylor expansion around θ of a gradient update (Achiam et al., 2019), as well as the Taylor expansion of pointwise loss differences (Liu et al., 2019b; Fort et al., 2019).

Interestingly, and as noted by works cited above, this quantity is intimately related to *generalization*. If the interference between two processes is positive, then updating θ using gradients from one process will positively impact the other. Such processes can take many forms, for example, J_1 being the loss on training data and J_2 the loss on *test data*, or J_1 and J_2 being the loss on two i.i.d. samples.

As a takeaway, and in terms of the previous section, **constructive interference is a sign of simple patterns being learned**, and is a form of generalization.

Deriving interference Consider an objective-based SGD update from $\nabla_\theta J$ using sample B (here A and B can be understood as samples, but in general they can be tasks, or even entire data distributions):

$$\theta' = \theta - \alpha \nabla_\theta J(B)$$

The effect of this update on the objective elsewhere, here at sample A, can be understood as the derivative of the loss elsewhere with respect to the learning rate, yielding the well-known gradient interference quantity ρ:

$$-\rho_{AB} = \frac{\partial J_{\theta'}(A)}{\partial \alpha} = -\frac{\partial J_{\theta'}(A)}{\partial \theta'}\frac{\partial \theta'}{\partial \alpha} \tag{3.7}$$

$$= -\nabla_{\theta'} J_{\theta'}(A)^T \nabla_\theta J_\theta(B) \tag{3.8}$$

$$\approx -\nabla_\theta J_\theta(A)^T \nabla_\theta J_\theta(B). \tag{3.9}$$

This quantity can also be obtained from the Taylor expansion of the loss difference at A after an update at B:

$$J_{\theta'(B)}(A) - J_\theta(A) \approx J_\theta(A) - J_\theta(A) + \nabla_\theta J(A)^T(\theta' - \theta) + O(||\theta' - \theta||^2) \tag{3.10}$$

$$\approx -\alpha \nabla_\theta J_\theta(A)^T \nabla_\theta J_\theta(B). \tag{3.11}$$

3.2.2 Temporal Difference changes Generalization Dynamics

What we would now like to claim is that in Temporal Difference Learning (TD), interference evolves differently during training than in supervised learning (SL). More specifically, we will show that **in TD learning lower interference correlates with a higher generalization gap while the opposite seems to be true in supervised learning**, where low interference correlates with a low generalization gap (the difference between test error and train error) when early stopping.

In supervised learning, there is a wealth of literature suggesting that SGD has a regularization effect (Hardt et al., 2016; Zhang et al., 2017; Keskar et al., 2017a, and references therein), pushing the parameters in flat highly-connected (Draxler et al., 2018; Garipov

et al., 2018) optimal regions of the loss landscape. It would be unsurprising for such regions of parameters to be on the threshold at the balance of bias and variance (in the traditional sense, referring here to the phase transition between generalizing and overfitting described in §3.1.2, i.e. when DNNs stop learning general patterns and start fitting dataset noise) and thus have low interference as well as a low generalization gap. Indeed, Fort et al. (2019) suggest that *stiffness*, the cosine similarity of gradients, drops but stays positive once a model starts overfitting. It is also suggested there that overfitting networks start having larger weights and thus larger gradients; this should result in the smallest ρ precisely before overfitting happens.

In RL, and in particular in TD-based RL methods, generalization has proven to be harder to achieve, or even to measure. This may be due to a multitude of factors, some also related to interference.

First, the evaluation methods of new algorithms in the recent union of neural networks and TD, despite an earlier recognition of the problem (Whiteson et al., 2011), often do not include generalization measures, perhaps leading to overfitting in algorithm space as well as solution space. This led to many works showing the brittleness of new TD methods (Machado et al., 2018; Farebrother et al., 2018; Packer et al., 2018; Witty et al., 2018), and works proposing to train on a distribution of environments (Zhang et al., 2018c; Cobbe et al., 2019; Justesen et al., 2018) in order to have proper training and test sets (Zhang et al., 2018a,b).

In TD methods, models also face a different optimization procedure where different components may be at odds with each other, leading to phenomena like the deadly triad (Sutton and Barto, 2018; Achiam et al., 2019) and leakage propagation (Penedones et al., 2018). In its purest version, the tabular bootstrapping of TD expects its targets to be fixed unless the target state is visited for an update; gradient updates create interference in unvisited target states, which breaks this assumption.

With most methods, from value-iteration to policy gradients, parameters are also faced with an inherently non-stationary optimization landscape. In particular for value-based methods, bootstrapping induces an asymmetric flow of information (from newly explored states to known states) which remains largely unexplored in deep learning literature. Such non-stationarity and asymmetry may help explain the success of sparse methods (Sutton, 1996; Liu et al., 2019a) that act more like tabular algorithms (with convergence at the cost of more updates).

Other works also underline the importance of interference. Riemer et al. (2018) show that by simply optimizing for interference across tasks via a naive meta-learning approach, one can improve RL performance. Interestingly, Nichol et al. (2018) also show how popular meta-learning methods implicitly also maximize interference (and thus constructive

updates). Considering that the meta-learning problem is inherently interested in generalization, this also suggests that increasing constructive interference should be beneficial.

3.2.2.1 Computing interference quantities

Comparing loss interference in the RL and SL case isn't necessarily indicative of the right trends, due to the fact that in most RL algorithms, the loss landscape itself evolves over time as the policy changes. Instead, we remark that loss interference, $\rho_{1,2} = \nabla_\theta J_1^T \nabla_\theta J_2$, can be decomposed as follows. Let J be a scalar loss, u and v some examples, and f the parameterized function's output:

$$\rho_{u,v} = \frac{\partial J(u)}{\partial f(u)} \frac{\partial f(u)}{\partial \theta}^T \frac{\partial f(v)}{\partial \theta} \frac{\partial J(v)}{\partial f(v)}. \tag{3.12}$$

While the partial derivative of the loss w.r.t. f may change as the loss changes, we find experimentally that the inner product of gradients of the output of f remains stable[1]. As such, we will also compute this quantity throughout, function interference, as it is more stable and reflects interference at the representational level rather than directly in relation to the loss function:

$$\bar{\rho}_{u,v} = \frac{\partial f(u)}{\partial \theta}^T \frac{\partial f(v)}{\partial \theta}. \tag{3.13}$$

For functions with more than one output in this work, e.g. a softmax classifier, we consider the output, $f(u)$, to be the max[2], e.g. the confidence of the argmax class.

3.2.2.2 Empirical Setup

For the generalization experiments of §3.2.2.3 we loosely follow the setup of Zhang et al. (2018a): we train RL agents in environments where the initial state is induced by a single random seed, allowing us to have proper training and test sets in the form of mutually exclusive seeds. In particular, to allow for closer comparisons between RL and SL, we compare classifiers trained on SVHN (Netzer et al., 2011) and CIFAR10 (Krizhevsky and Hinton, 2009) to agents that learn to progressively explore a masked image (from those datasets) while attempting to classify it. The random seed in both cases is the index of the example in the training or test set.

More specifically, agents start by observing only the center, an 8×8 window of the current image. At each time-step they can choose from 4 movement actions, moving the

[1] Although gradients do not always converge to 0, at convergence the parameters themselves tend to *wiggle around* a minima, and as such do not affect the *function* and its derivatives that much.

[2] This avoids computing the (expensive) Jacobian, we also find that this simplification accurately reflects the same trends experimentally.

observation window by 8 pixels and revealing more of the image, as well as choose from 10 classification actions. The episode ends upon a correct classification or after 20 steps.

We train both RL and SL models with the same architectures, and train RL agents with a Double DQN objective (van Hasselt et al., 2016). We also train REINFORCE (Williams, 1992c) agents as a test to entirely remove dependence on value estimation and have a pure Policy Gradient (PG) method.

DDQN agents maintain two sets of parameters, θ and $\bar{\theta}$, that estimate the action-value function Q. θ is trained to minimize

$$\mathcal{L}_{QL} = [Q_\theta(S_t, A_t) - (R_t + \gamma \max_a Q_{\bar{\theta}}(S_{t+1}, a))]^2 \tag{3.14}$$

on trajectories $S_0, A_0, R_0, ...$ sampled from the ϵ-greedy policy, and $\bar{\theta}$ is updated after each update of θ to be its exponential moving average. REINFORCE agents maintain one set of parameters θ that estimate the policy π, and are modified according to the gradient:

$$\nabla_\theta G(S_t) = G(S_t) \nabla_\theta \log \pi(A_t|S_t), \tag{3.15}$$

on trajectories sampled from π. In both cases the reward given to the agent is 1 on a correct classification and -0.1 for moving the observation window or an incorrect classification. Supervised models are trained to maximize the log-likelihood of the labels.

As much of the existing deep learning literature on generalization focuses on classifiers, but estimating value functions is arguably closer to regression, we include two regression experiments using SARCOS (Vijayakumar and Schaal, 2000) and the California Housing dataset (Pace and Barry, 1997).

Finally, for the interactive environment experiments of §3.2.2.4 and §3.2.3, we investigate some metrics on the popular Atari environment (Bellemare et al., 2013) by training DQN (Mnih et al., 2013) agents, with the stochastic setup recommended by Machado et al. (2018), and by performing policy evaluation with the Q-Learning and TD(λ) objectives. To generate interesting trajectories for policy evaluation, we run an "expert" agent pre-trained with Rainbow (Hessel et al., 2018); we denote $\mathcal{D}^*$ a dataset of transitions obtained with this agent, and θ^* the parameters after training that agent.

We measure correlations throughout with Pearson's r:

$$r_{xy} = \frac{\sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^n (x_i - \bar{x})^2}\sqrt{\sum_{i=1}^n (y_i - \bar{y})^2}} \tag{3.16}$$

which is a measure, between -1 and 1, of the linear correlation between two random variables X, Y. All architectural details and hyperparameter ranges are listed in 3.2.4.2. All code is available at `https://github.com/bengioe/interference_and_generalization_in_td/`.

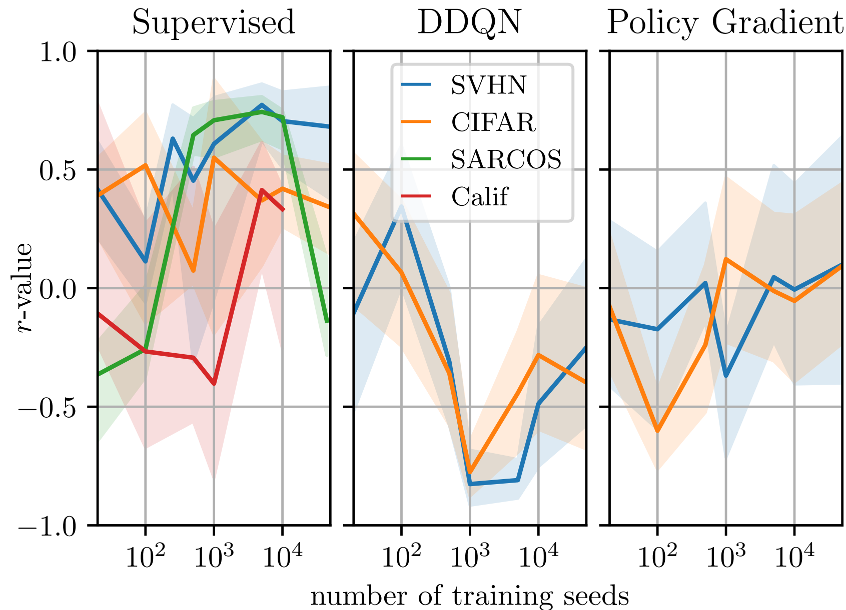


Figure 3.4: Correlation coefficient r between the (log) function interference $\bar{\rho}$ and the generalization gap, as a function of training set size; shaded regions are bootstrapped 90% confidence intervals. We see different trends for value-based experiments (middle) than for supervised (left) and PG experiments (right). For classification, these methods clearly have different effect on interference even though they roughly approximate the same function.

3.2.2.3 Natural data generalization experiments

To measure interference in the overparameterized regime and still be able to run many experiments to obtain trends, we instead reduce the number of training samples while also varying capacity (number of hidden units and layers) with smaller-than-state-of-the-art but reasonable architectures. Since in RL we could consider one trajectory to be a single sample, we generalize this notion into the notion of "training seeds"–one per sample in supervised learning and one per trajectory in RL.

First, in Figure 3.4 for each training set size, we measure the correlation between interference and the generalization gap. We see that, after being given sufficient amounts of data, TD methods tend to have a strong negative correlation, while classification methods tend to have positive correlation.

Regression has similar but less consistent results; SARCOS has a high correlation peak when there starts being enough data, albeit shows no correlation at all when all 44k training examples are given (the generalization gap is then almost 0 for all hyperparameters); on the other hand the California dataset only shows positive correlation when most or all of the dataset is given. The trends for PG SVHN and CIFAR show no strong correlations (we note that $|r| < 0.3$ is normally considered to be a weak correlation; Cohen, 2013) except

for PG CIFAR at 100 training seeds, with $r = -.60$.

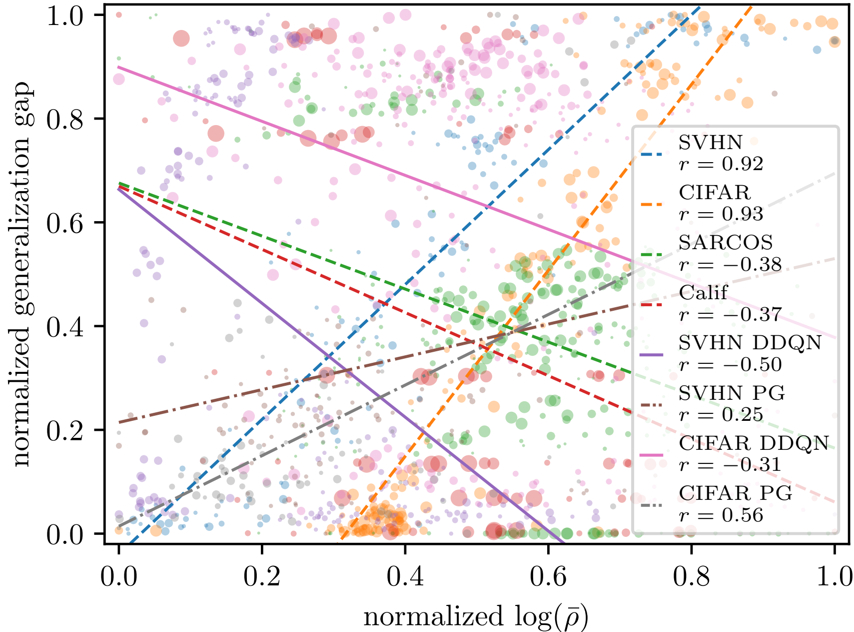


Figure 3.5: Generalization gap vs interference $\bar{\rho}$ for all runs. Larger circles represent larger capacity models. Here value-based methods seem to be behaving like regression methods.

Second, in Figure 3.5, we plot the generalization gap against interference $\bar{\rho}$ for every experiment (normalized for comparison). We then draw the linear regression for each experiment over all training set sizes and capacities. For both classification tasks, interference is strongly correlated ($r > 0.9$) with the generalization gap, and also is to a lesser extent for the PG experiments. For all other experiments, regression and value-based, the correlation is instead negative, albeit low enough that a clear trend cannot be extracted. Note that the generalization gap itself is almost entirely driven by the training set size first ($r < -0.91$ for all experiments except PG, where r is slightly higher, as seen in Figure 3.6).

The combination of these results tells us that (1) interference evolves differently in TD than in SL, (2) interference for TD has some similarities with regression, as well as a different characterization of memorization: **in classification, low-interference solutions tend to generalize, while in TD, low-interference solutions often memorize**. In regression, this seems only true for a fixed quantity of data.

3.2.2.4 Interference in Atari domains

The Arcade Learning Environment (Bellemare et al., 2013), comprised of Atari games, has been a standard Deep RL benchmark in the recent past (Mnih et al., 2013; Bellemare et al., 2017; Kapturowski et al., 2019). We once again revisit this benchmark to provide additional

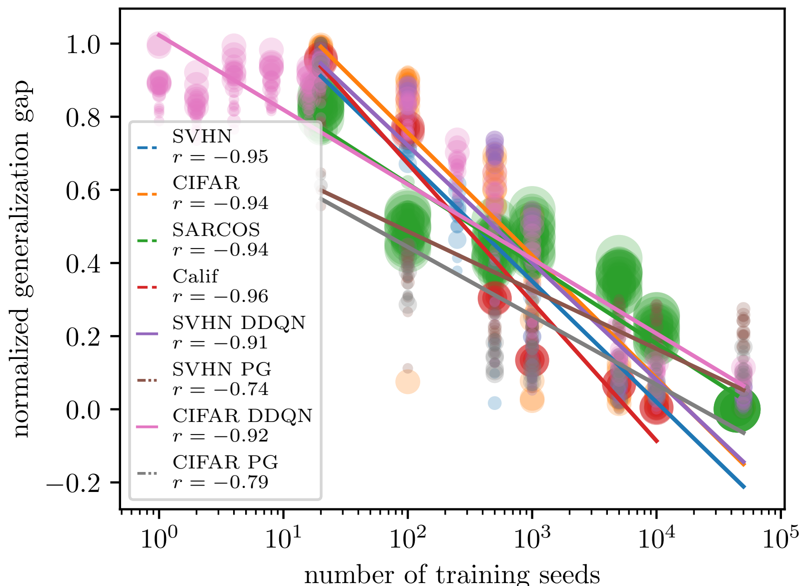


Figure 3.6: Generalization gap vs number of training seeds. The size of each circle (which represents a single experiment) is proportional to the number of hidden units.

evidence of the memorization-like behaviours of value-based methods on these domains. Understanding the source of these behaviours is important, as presumably algorithms may be able to learn generalizing agents from the same data. Additionally, such low-interference memorization behaviours are not conducive to sample efficiency, which even in an environment like Atari, could be improved.

Recall that interference is a first order Taylor expansion of the pointwise loss difference, $J_{\theta'} - J_\theta$. Evaluating such a loss difference is more convenient to do on a large scale and for many runs, as it does not require computing individual gradients. In this section, we evaluate the expected TD loss difference for several different training objectives, a set of supervised objectives, the Q-Learning objective applied first as policy evaluation (learning from a replay buffer of expert trajectories) and then as a control (learning to play from scratch) objective, and the TD(λ) objective applied on policy evaluation. Experiments are ran on MsPacman, Asterix, and Seaquest for 10 runs each. Results are averaged over these three environments (they have similar magnitudes and variance). Learning rates are kept constant, they affect the magnitude but not the shape of these curves. We use 10M steps in the control setting, and 500k steps otherwise.

We first use the following 3 supervised objectives to train models using $\mathcal{D}^*$ as a dataset

and Q_{θ^*} as a *distillation* target:

$$\mathcal{L}_{MC}(s,a) = (Q_\theta(s,a) - G^{(\mathcal{D}^*)}(s))^2$$
$$\mathcal{L}_{reg}(s,a) = (Q_\theta(s,a) - Q_{\theta^*}(s,a))^2$$
$$\mathcal{L}_{TD^*}(s,a,r,s') = (Q_\theta(s,a) - (r + \gamma \max_{a'} Q_{\theta^*}(s',a')))^2$$

and measure the difference in pointwise TD loss ($\mathcal{L}_{QL}$) for states *surrounding* the state used for the update (i.e. states with a temporal offset of ±30 in the replay buffer trajectories), shown in Figure 3.7.

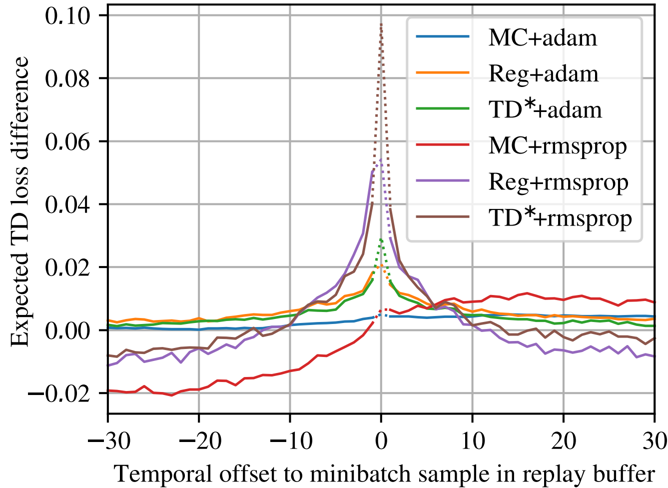


Figure 3.7: Regression on Atari: loss difference as a function of temporal offset in the replay buffer from the update sample. We use dotted lines at 0 offset to emphasize that the corresponding state was used for the update. The curve around 0 is indicative of the constructive interference of the TD and regression objectives.

There, we see that curves tend to be positive around $x = 0$ (the sample used in the update), especially from indices -10 to 10, showing that **constructive interference is possible** when learning to approximate Q^* with this data. Since Q_{θ^*} is a good approximation, we expect that $Q_{\theta^*}(s,a) \approx (r + \gamma \max_{a'} Q_{\theta^*}(s',a'))$, so $\mathcal{L}_{reg}$ and $\mathcal{L}_{TD^*}$ have similar targets and we expect them to have similar behaviours. Indeed, their curves mostly overlap.

Next, we again measure the difference in pointwise loss for surrounding states. We train control agents and policy evaluation (or *Batch Q*) agents with the Q-Learning loss:

$$\mathcal{L}_{QL} = [Q_\theta(S_t, A_t) - (R_t + \gamma \max_a Q_\theta(S_{t+1}, a))]^2. \tag{3.17}$$

We show the results in Figure 3.8. Compared to the regressions in Figure 3.7, the pointwise difference is more than an order of magnitude smaller, and drops off even faster

when going away from $x = 0$. This suggests a low interference, and a low update propagation. For certain optimizers, here RMSProp (Hinton et al., 2012) and SGD, this effect is even slightly negative. We believe this difference may be linked to momentum (note the difference with Adam (Kingma and Ba, 2015) and Momentum-SGD), which might dampen some of the negative effects of TD on interference (further discussed in §3.2.3.3).

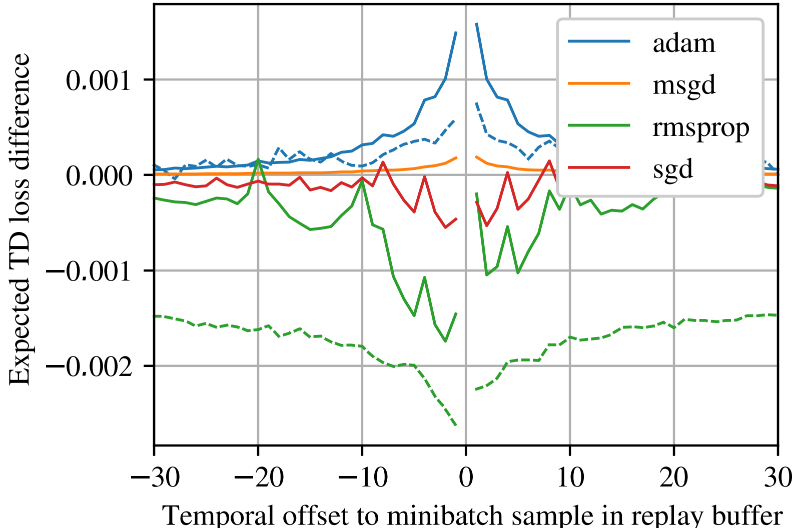


Figure 3.8: TD Learning on Atari: loss difference as a function of offset in the replay buffer of the update sample. Full lines represent Q-Learning control experiments, while dashed lines represent policy evaluation with a Q-Learning objective. We exclude $x = 0$ for clarity, as it has a high value (see Figure 3.9). Compared to regression, the magnitude of the gain is much smaller.

Interestingly, while Q-Learning does not have as strong a gain as the regressions from Figure 3.7, it has a larger gain than policy evaluation. This may have several causes, and we investigate two.

First, we hypothsize that due the initial random exploratory policy, the DNN initially sees little data variety, and may be able to capture a minimal set of factors of variation; then, upon seeing new states, the extracted features are forced to be mapped onto those factors of variation, improving them, leading to a natural curriculum. By looking at the *singular values* of a decomposition of the last hidden layer's weight matrix after 100k steps, we do find that there is a *consistently larger spread* in the policy evaluation case than the control case, suggesting that in the control case fewer factors are initially captured. This effect diminishes as training progresses.

Figure 3.10 shows the spread of singular values after 100k minibatch updates on MsPac-

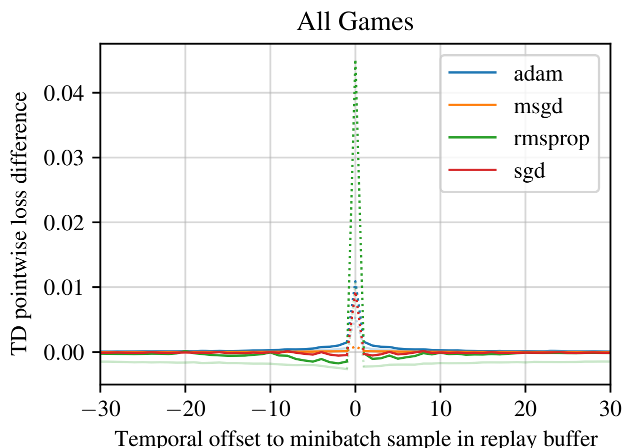


Figure 3.9: Reproduction of Figure 3.8 including $x = 0$. RMSprop has a surprisingly large expected gain at $x = 0$, but a negative gain around $x = 0$, suggesting that RMSprop enables memorization more than Adam.

man for the Q-Learning objective and Adam/RMSProp. The difference between the control case and policy evaluation supports our hypothesis that policy evaluation initially captures more factors of variation. It remains unclear if the effect of the control case initially having fewer captured factors of variation leads to a form of feature curriculum.

Note that current literature suggests that having fewer large singular values in the weight matrices of neural networks is a sign of generalization *in classifiers*, see in particular Oymak et al. (2019), as well as Morcos et al. (2018) and Raghu et al. (2017). It is not clear whether this holds for regression, nor in our case for value functions. Interestingly all runs, even for TD(λ), have a dramatic cutoff in singular values after about the 200th SV, suggesting that there may be in this order of magnitude many underlying factors in MsPacman, and that by changing the objective and the data distribution, a DNN may be able to capture them faster or slower.

Second, having run for 10M steps, control models could have been trained on more data and thus be forced to generalize better; this turns out **not** to be the case, as measuring the same quantities for only the first 500k steps yields very similar magnitudes. In other words, after a few initial epochs, function interference remains constant: Figure 3.11 shows the evolution of TD pointwise loss difference during training; in relation to previous figures like Figure 3.8, the y axis is now Figure 3.8's x axis – the temporal offset to the update sample in the replay buffer, the y axis is now training time, and the color is now Figure 3.8's y axis – the magnitude of the TD gain.

Interestingly, these results are consistent with those of Agarwal et al. (2019), who study

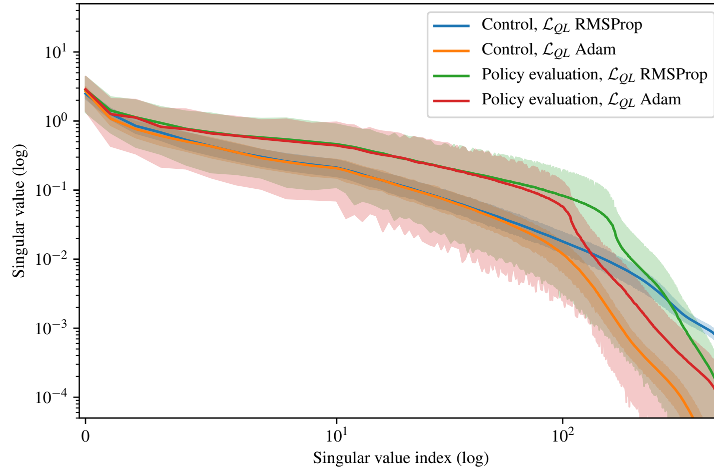


Figure 3.10: Spread of singular values of the last hidden layer's weight matrix after 100k iterations. Despite having seen the same amount of data, the control experiments generally have seen fewer unique states, which may explain the observed difference. Shaded regions show bootstrapped 95% confidence intervals.

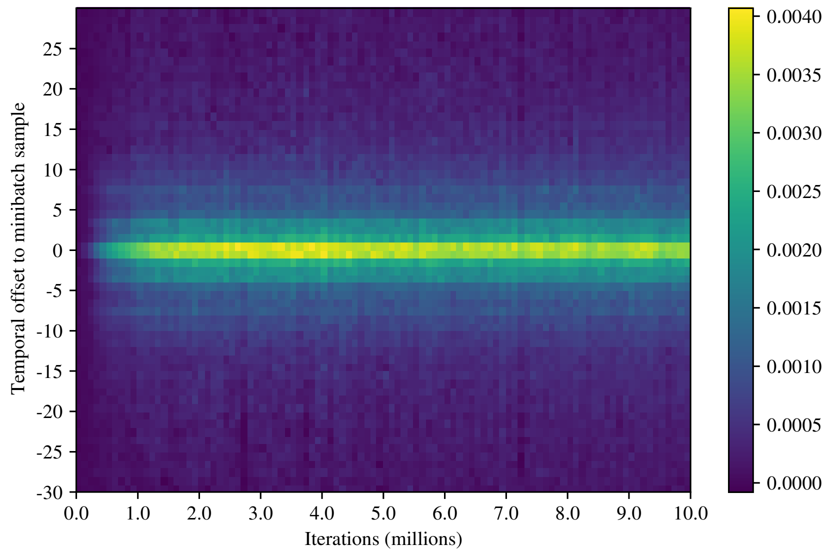


Figure 3.11: Evolution of TD pointwise loss difference, during training. Control experiment with Adam, MsPacman, averaged over 10 runs. Note that index 0 is excluded as its magnitude would be too large and dim all other values.

off-policy learning. Among many other results, Agarwal et al. (2019) find that off-policy-retraining a DQN model (i.e. Batch Q-Learning) on another DQN agent's lifetime set of trajectories yields much worse performance. This is consistent with our results showing more constructive interference in control than in policy evaluation, and suggests that the order in which data is presented may matter when bootstrapping is used.

3.2.3 The Instability of Deep Temporal Difference Learning

In §3.2.2.3 and §3.2.2.4 we have shown that TD-based methods induce different interference dynamics.

We first showed that these manifest in terms of train-test generalization: in §3.2.2.3, we could link the test-time performance of RL agents to the interference between their training examples, and showed that this interference was different than for SL models.

We then showed in §3.2.2.4 that in control domains such as Atari, this difference between SL and RL manifests in terms of a highly reduced constructive interference, or in other words state-to-state generalization. This possibly causes learning to be much slower.

We will now attempt to dissect where this phenomenon comes from, and find that it is rooted in the instability of optimization with a Temporal Difference objective.

3.2.3.1 TD(λ) and bootstrapping

A central hypothesis of this work is that bootstrapping causes instability in interference, causing it to become small and causing models to memorize more. Here we perform policy evaluation on $\mathcal{D}^*$ with TD(λ). TD(λ) is by design a way to trade-off between bias and variance in the target by trading off between few-step bootstrapped targets and long-term bootstrapped targets which approach Monte-Carlo returns. In other words, TD(λ) allows us to diminish reliance on bootstrapping.

Intuitively, **TD(λ)** trades off between the unbiased target $G(S_t)$ and the biased TD(0) target (biased due to relying on the estimated $V(S_{t+1})$) by trading off between different intermediate targets, one for each step of a trajectory. More specifically, this trade off is achieved by using a weighted averaging of future targets called a λ-return (Sutton, 1988; Munos et al., 2016):

$$G^\lambda(S_t) = (1-\lambda)\sum\nolimits_{n=1}^{\infty}\lambda^{n-1}G^n(S_t) \tag{3.18}$$

$$G^n(S_t) = \gamma^n V(S_{t+n}) + \sum\nolimits_{j=0}^{n-1}\gamma^j R(S_{t+j})$$

$$\mathcal{L}_{TD(\lambda)}(S_t) = (V_\theta(S_t) - G^\lambda(S_t))^2, \tag{3.19}$$

for $\lambda \in (0, 1]$. Note that the return depends implicitly on the trajectory and the actions followed from S_t. When $\lambda = 0$, the loss is simply $(V_\theta(S_t) - (R_t + \gamma V_\theta(S_{t+1})))^2$, leading to the TD(0) algorithm (Sutton, 1988). Higher values of λ reduce the dependency on V (i.e. reduce the bias) but increase the dependency on G, which is a high-variance random variable.

This trade-off is especially manifest when measuring the *stiffness* of gradients (cosine similarity) as a function of temporal offset, as shown in Figure 3.12. There we see that the

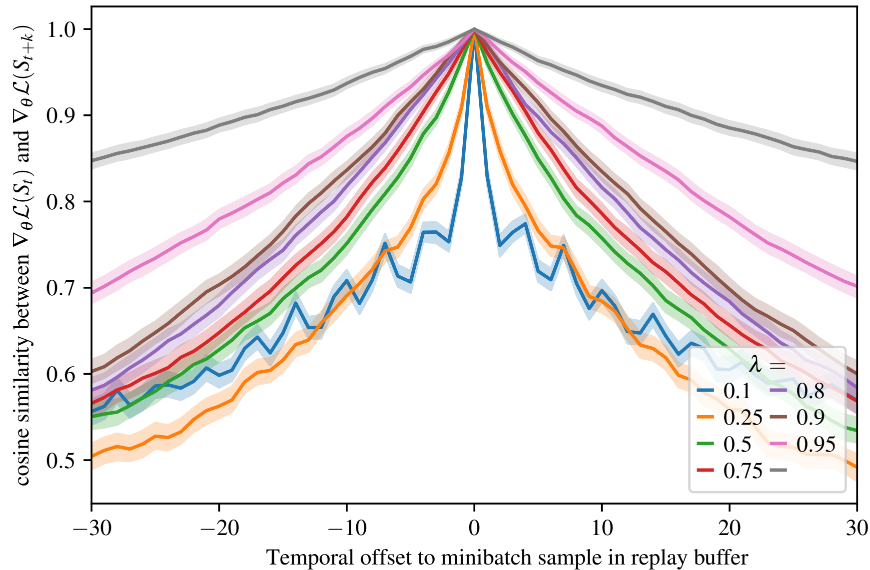


Figure 3.12: Cosine similarity between gradients at S_t (offset $x = 0$) and the gradients at the neighbouring states in the replay buffer (MsPacman). As λ increases, so does the temporal coherence of the gradients.

closer λ is to 1, the more gradients are similar around an update sample, suggesting that diminishing reliance on bootstrapping reduces the effect of TD inducing low-interference memorizing parameterizations.

Note that this increase in similarity between gradients is also accompanied by an increase in pointwise loss difference (shown in Figure 3.13), surpassing that of Q-Learning (Figure 3.8) in magnitude. This suggests that TD(λ) offers more coherent targets that allow models to learn faster, for sufficiently high values of λ.

Figure 3.14 shows the spread of singular values after 500k minibatch updates for TD(λ). Interestingly, larger λ values yield larger singular values and a wider distribution. Presumably, TD(λ) having a less biased objective allows the parameters to capture all the factors of variation faster rather than to rely on bootstrapping to gradually learn them.

Here, by simply changing the target in the objective of Temporal Difference learning, we've shown a notable improvement in constructive interference, suggesting that the TD(0) target coupled with standard optimization methods and deep neural networks induces the "bad" parameterizations we've discussed in previous sections. Since we know that this target is biased, let's now look at *how* it is biased and what this implies.

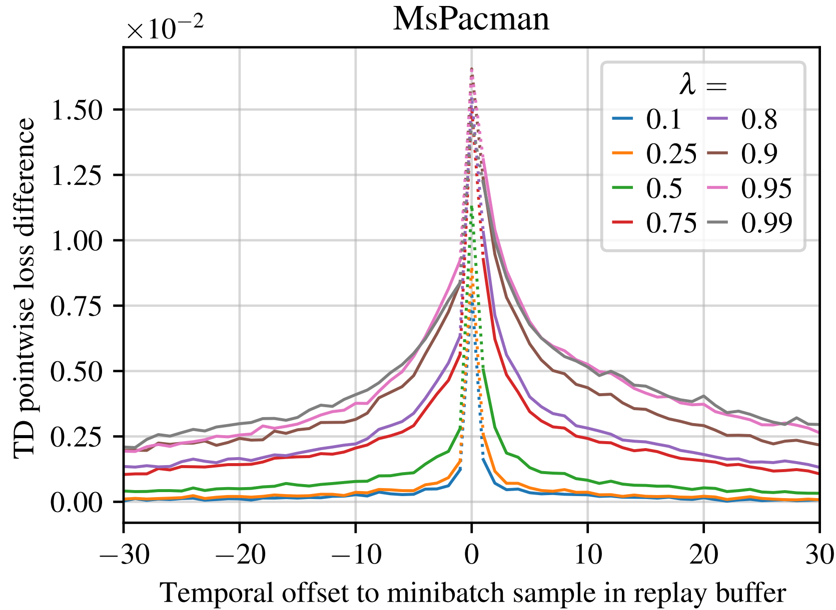


Figure 3.13: Evolution of TD pointwise loss difference, as a function of λ in TD(λ). Notice the asymmetry around 0.

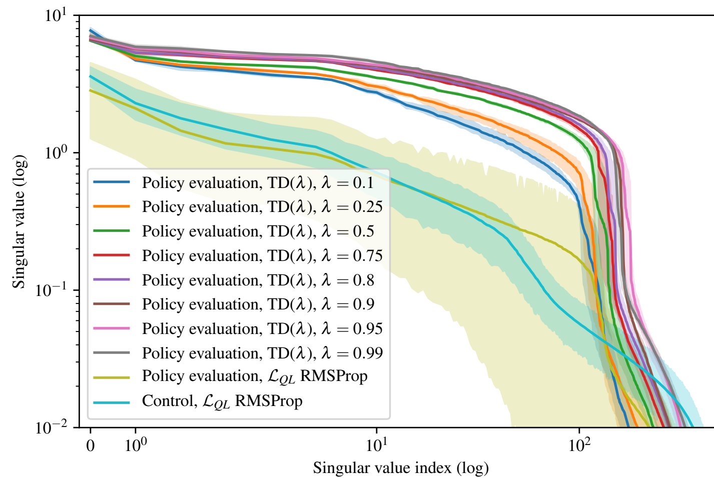


Figure 3.14: Spread of singular values after 500k iterations. Shaded regions show bootstrapped 95% confidence intervals.

3.2.3.2 The high variance of target-bias in TD(0)

In TD(0), the current target for any state depends on the prediction made at the next state. The difference between that prediction and the true value function makes the target a biased estimator when bootstrapping is in progress and information flows from newly visited states to seen states.

This "bootstrap bias" itself depends on a function approximator which has its own

bias-variance trade-off (in the classical sense). For a high-variance approximator, this bootstrap bias might be inconsistent, making the value function alternate between being underestimated and being overestimated, which is problematic in particular for nearby states[3]. In such a case, a gradient descent procedure cannot truly take advantage of the constructive interference between gradients.

Indeed, recall that in the case of a regression, interference can be decomposed as:

$$\rho_{x,y} = \frac{\partial J(x)}{\partial f(x)} \frac{\partial f(x)}{\partial \theta}^T \frac{\partial f(y)}{\partial \theta} \frac{\partial J(y)}{\partial f(y)},$$

which for the TD error $\delta_x = V(x) - (r(x) + \gamma V(x'))$ with x' some successor of x, can be rewritten as:

$$\rho_{x,y} = \delta_x \delta_y \nabla_\theta V(x)^T \nabla_\theta V(y).$$

If x and y are nearby states, in some smooth high-dimensional input space (e.g. Atari) they are likely to be close in input space and thus to have a positive function interference $\nabla_\theta V(x)^T \nabla_\theta V(y)$. If the signs of δ_x and δ_y are different, then an update at x will increase the loss at y. As such, we measure the variance of the sign of the TD error along small windows (of length 5 here) in trajectories as a proxy of this local target incoherence.

We observe this at play in Figure 3.15, which shows interference and rewards as a function of sign variance for a DQN agent trained on MsPacman. As predicted, parameterizations with a large $\bar{\rho}$ and a large sign variance perform much worse. We note that this effect can be lessened by using a much smaller learning rate than is normal, but this comes at the cost of having to perform more updates for a similar performance (in fact, presumably because of reduced instability, performance is slightly better, but only towards the end of training; runs with a normal α plateau halfway through training).

Interestingly, parameterizations with large $\bar{\rho}$ *generally do* have a large sign variance ($r = 0.71$) in the experiment of Figure 3.15. Indeed, we believe that the evolution of interference in TD methods may be linked to sign variance, the two compounding together, and may explain this trend.

These results are consistent with the improvements obtained by Thodoroff et al. (2018), who force a temporal smoothing of the value function through convex combinations of sequences of values which likely reduces sign variance. These results are also consistent with those of Anschel et al. (2017) and Agarwal et al. (2019) who obtain improvements by training ensembles of value functions. Such ensembles should partly reduce the sign

[3] Consider these two sequences of predictions of V: $[1, 2, 3, 4, 5, 6]$ and $[1, 2, 1, 2, 1, 2]$. Suppose no rewards, $\gamma = 1$, and a function interference ($\bar{\rho}$) close to 1 for illustration, both these sequences have the same average TD(0) error, 1, yet the second sequence will cause any TD(0) update at one of the states to only correctly update half of the values.

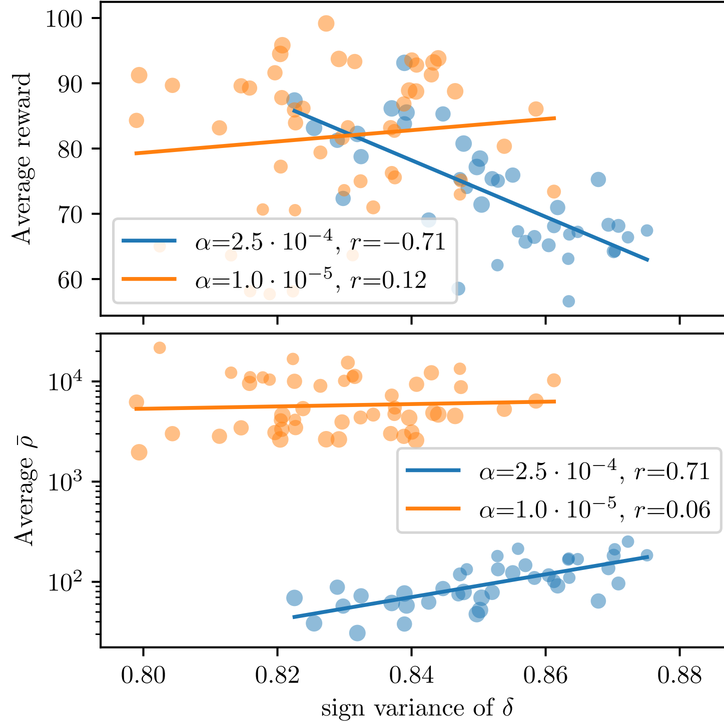


Figure 3.15: Top, average reward after training as a function of the sign variance for different learning rates (α) and number of hidden units (size of markers). We can see that by using a much smaller learning rate than normal, the biasing effect of TD is lessened, at the cost of many more updates. Bottom, average function interference $\bar{\rho}$ after training. We see that, as predicted, parameterizations with large $\bar{\rho}$ and a large sign variance perform much worse (note that the x-axis of both plots are aligned, allowing for an easy reward/interference comparison).

variance of δ, as bias due to initialization should average to a small value, making targets more temporally consistent.

Finally, note that in regression, this problem may eventually go away as parameters converge. Instead, in TD(0), especially when making use of a frozen target, this problem simply compounds with time and with every update. In what follows we consider this problem analytically.

3.2.3.3 Understanding the evolution of interference

Here we attempt to provide some insights into how interference evolves differently in classification, regression, and TD learning. For detailed derivations see §3.2.4.1.

Recall that the interference ρ can be obtained by the negative of the derivative of the

loss $J(A)$ after some update using B w.r.t. the learning rate α, i.e.

$$\theta' = \theta - \alpha \nabla_\theta J(B) \tag{3.20}$$

$$\rho_{AB} = -\partial J_{\theta'}(A)/\partial\alpha = \nabla_{\theta'} J(A) \cdot \nabla_\theta J(B) \tag{3.21}$$

$$\approx \nabla_\theta J(A) \cdot \nabla_\theta J(B). \tag{3.22}$$

The last step being a simplification as $\theta \approx \theta'$.

To try to understand how this quantity evolves, we can simply take the derivative of ρ (and $\bar{\rho}$) w.r.t. α but evaluated at θ', that is, $\rho'_{AB} = \partial(\nabla_{\theta'} J(A) \cdot \nabla_{\theta'} J(B))/\partial\alpha$. In the general case, we obtain (assuming $\theta \approx \theta'$, we omit the θ subscript and subscript A and B for brevity):

$$\rho'_{AB} = -(\nabla J_B^T H_A + \nabla J_A^T H_B)\nabla J_B \tag{3.23}$$

$$\bar{\rho}'_{AB} = -(\nabla f_B^T \bar{H}_A + \nabla f_A^T \bar{H}_B)\nabla J_B \tag{3.24}$$

where $H_A = \nabla^2_\theta J(A;\theta)$, $\bar{H}_A = \nabla^2_\theta f(A;\theta)$ are Hessians, $f_A = f(A;\theta)$.

Interpreting this quantity is non-trivial, but consider $\nabla f_A^T \bar{H}_B \nabla J_B$; parameters which make f_A change, which have high curvature at B (e.g. parameters that are not stuck in a saddle point or a minima at B), and which change the loss at B will contribute to change ρ. Understanding the sign of this change requires a few more assumptions.

Because neural networks are somewhat smooth (they are Lipschitz continuous, although their Lipschitz constant might be very large, see Scaman and Virmaux (2018)), it is likely for examples that are close in input space *and* target space to have enough gradient and curvature similarities to increase their corresponding interference, while examples that are not similar would decrease their interference. Such an interpretation is compatible with our results, as well as those of Fort et al. (2019) who find that *stiffness* (cosine similarity of gradients) is mostly positive only for examples that are in the same class.

Indeed, notice that for a given softmax prediction σ, for A and B of different classes y_A, y_B, the sign of the partial derivative at $\sigma_{y_A}(A)$ will be the opposite of that of $\sigma_{y_A}(B)$. Since gradients are multiplicative in neural networks, this will flip the sign of all corresponding gradients related to σ_{y_A}, causing a mismatch with curvature, and a decrease in interference. Thus the distribution of targets and the loss has a large role to play in aligning gradients, possibly just as much as the input space structure.

We can also measure ρ' to get an idea of its distribution. For a randomly initialized neural network, assuming a normally distributed input and loss, we find that it does not appear to be 0 mean. While the median is close to 0, but consistently negative, the distribution seems heavy-tailed with a slightly larger negative tail, making the negative mean further away from 0 than the median. In what follows we decompose ρ' to get some additional insights.

In the case of regression, $J_A = 1/2(f_A - y_A)^2$, $\delta_A = f_A - y_A$, we get that:

$$\begin{aligned}\rho'_{reg;AB} = &-\bar{\rho}^2_{AB}\delta^2_B - 2\delta_A\delta_B\bar{\rho}_{AB}\bar{\rho}_{BB} \\ &- \delta_A\delta^2_B\nabla f_B(\bar{H}_A\nabla f_B + \bar{H}_B\nabla f_A)\end{aligned} \tag{3.25}$$

Another interesting quantity is the evolution of ρ when J is a TD loss if we assume that the bootstrap target also changes after a weight update. With the $\theta \approx \theta'$ simplification, $\delta_A = V_A - (r + \gamma V_{A'})$ the TD error at A, A' some successor of A, we get:

$$\begin{aligned}\rho'_{TD;AB} = &-\delta^2_B\bar{\rho}_{AB}(\bar{\rho}_{AB} - \gamma\bar{\rho}_{A'B}) \\ &- \delta_A\delta_B\bar{\rho}_{AB}(\bar{\rho}_{BB} - \gamma\bar{\rho}_{B'B}) \\ &- \delta_A\delta^2_B\nabla f_B(\bar{H}_A\nabla f_B + \bar{H}_B\nabla f_A)\end{aligned} \tag{3.26}$$

Again considering the smoothness of neural networks, if A and B are similar, but happen to have opposite δ signs, their interference will decrease. Such a scenario is likely for high-capacity high-variance function approximators, and is possibly **compounded by the evolving loss landscape**. As the loss changes–both prediction and target depend on a changing θ–it evolves imperfectly, and there are bound to be many pairs of nearby states where only one of the δs flips signs, causing gradient misalignments. This would be consistent with our finding that higher-capacity neural networks have a smaller interference in TD experiments (see Figure 3.16) while the reverse is observed in classification.

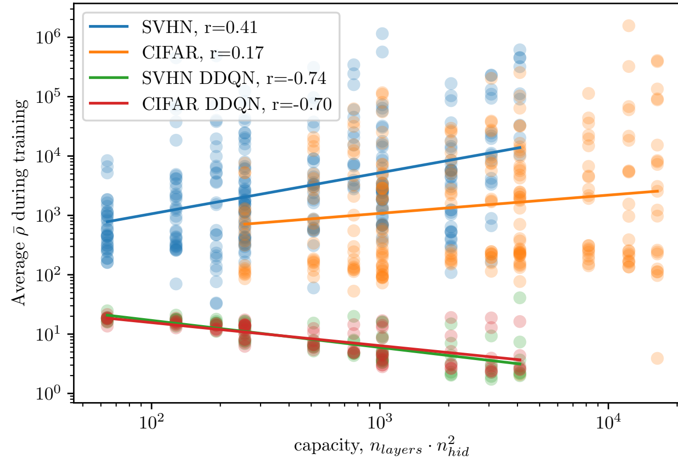


Figure 3.16: Average function interference during training as a function of capacity. TD methods and classifiers have very different trends.

We now separately measure the three additive terms of ρ'_{reg} and ρ'_{TD}, which we refer to as $\rho' = -r_1 - r_2 - r_3$, in the same order in which they appear in (3.25) and (3.26).

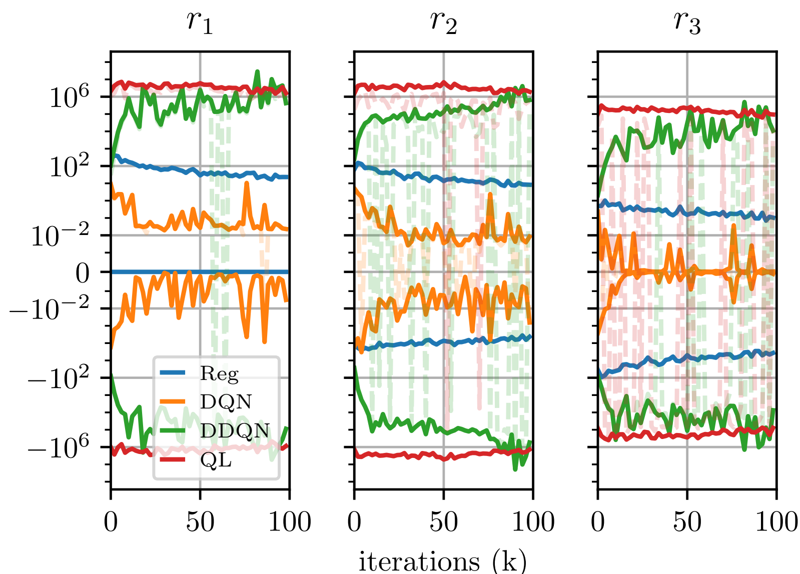


Figure 3.17: r_1, r_2, r_3 for ρ'_{reg} (Reg) and ρ'_{TD} (DQN, DDQN, QL) measured early in training. The transparent dashed lines are the mean r_i, averaged over 1024 (32×32) sample pairs, averaged over 3 runs. The full lines above and below 0 are the average of the positive and negative samples of r_i respectively. These lines show the relative magnitudes of each part: in general, positive samples dominate for r_1, r_2 varies a lot between positive and negative for TD, while r_3 is mostly negative with some variance for TD.

We measure these terms in four scenarios, using a MsPacman expert replay buffer. We regress to $Q_{\theta*}$ (measuring ρ'_{reg}), and run policy evaluation with three different targets (measuring ρ'_{TD}). In DQN, the target $Q_{\bar{\theta}}$ is a frozen network updated every 10k iterations; in DDQN the target is updated with an exponential moving average rule, $\bar{\theta} = (1-\tau)\bar{\theta}+\tau\theta$, with $\tau = 0.01$; in QL the target is the model itself Q_θ as assumed in (3.26). This is shown in Figure 3.17. We see that in regression r_1 and r_2 are positive much more often than they are negative, while in TD methods, the positive samples tend to dominate but the proportion of negative samples is much larger, especially for r_2, which contains a $\delta_A\delta_B$ product. We see that r_3 tends to have a smaller magnitude than other terms for TD methods, and is negative on average.

These results suggest that interference evolves poorly when applying SGD to variants of the TD objective. On the other hand, momentum SGD is the optimizer of choice in the literature and yields better performing agents. Could it have a beneficial effect?

Momentum SGD has the following updates, $\beta \in [0, 1)$:

$$\mu_t = (1-\beta)\nabla_\theta J_B + \beta\mu_{t-1} \tag{3.27}$$

$$\theta' = \theta - \alpha(\beta\mu_{t-1} + (1-\beta)\nabla_\theta J_B) \tag{3.28}$$

yielding the following quantities:

$$\rho_{\mu;AB} = (1-\beta)\nabla_{\theta'} J_A \cdot \nabla_\theta J_B + \beta\nabla_{\theta'} J_A \cdot \mu_{t-1} \tag{3.29}$$

$$\rho'_{\mu;AB} = -(1-\beta)\rho'_{AB} - \beta\nabla J_B H_A \mu_{t-1} \tag{3.30}$$

Note that the first term of $\rho'_{\mu;AB}$ is simply eq. (3.23) times $1-\beta$. The second term is more interesting, and presumably larger as β is usually close to 1. It indicates that for interference to change, the curvature at A and the gradient at B need to be aligned with μ, the moving average of gradients. As such, the evolution of interference may be driven more by the random (due to the stochasticity of SGD) alignment of the gradients with μ, which should be stable over time since μ changes slowly, than by the (high-variance) alignment of curvature at A and gradient at B.

As such, momentum should lower the variance of ρ' and dampen the evolution of interference when it is high-variance, possibly including dampening the negative effects of interference in TD. Unfortunately, applying momentum to bootstrapping updates may be not be fully principled, as we will discuss later in this thesis.

Overall, these results and derivations suggest that TD methods do not have a stable evolution of interference, at least not when *naively applying SGD* to the TD objective. In some sense, this is unsurprising given that Temporal Difference as it is applied here is *not* a "gradient" method (although such methods do exist, as we will discuss later). Such findings are the central motivation to a future part of this thesis, §4.1, concerned with optimization and TD.

3.2.4 Derivations and Hyperparameters

3.2.4.1 Second order quantities of interference

The derivative of ρ w.r.t. α, or second order derivative of $J_{\theta'}$ w.r.t. α is:

$$\frac{\partial^2 J_{\theta'}(A)}{\partial\alpha^2} = -\frac{\partial}{\partial\alpha}\nabla_{\theta'} J_A^T \nabla_\theta J_B \tag{3.31}$$

$$= -\left(\frac{\partial(\nabla_{\theta'} J(A))}{\partial\theta'}\frac{\partial\theta'}{\partial\alpha}\right)^T \nabla_\theta J_B \tag{3.32}$$

$$= -(-\nabla^2_{\theta'} J_A \nabla_\theta J_B)^T \nabla_\theta J_B \tag{3.33}$$

$$\approx \nabla J_B^T H_A \nabla J_B \tag{3.34}$$

assuming $\theta \approx \theta'$ in the last step, and where $H_A = \nabla^2_\theta J_A$ is the Hessian. Again the only approximation here is $\theta \approx \theta'$.

While this quantity is interesting, it is in a sense missing a part: what happens to the interference itself after an update? At **both** A and B at θ'?

$$\rho'_{AB} = \frac{\partial}{\partial\alpha} \nabla_{\theta'} J_A^T \nabla_{\theta'} J_B \tag{3.35}$$
$$= (\frac{\partial(\nabla_{\theta'} J_A)}{\partial\theta'} \frac{\partial\theta'}{\partial\alpha})^T \nabla_\theta J_B + \nabla_{\theta'} J_A^T (\frac{\partial(\nabla_{\theta'} J_B)}{\partial\theta'} \frac{\partial\theta'}{\partial\alpha}) \tag{3.36}$$
$$= (-\nabla^2_{\theta'} J_A \nabla_\theta J_B)^T \nabla_\theta J_B + \nabla_{\theta'} J_A^T (-\nabla^2_{\theta'} J_B \nabla_\theta J_B) \tag{3.37}$$
$$\approx -\nabla J_B^T H_A \nabla J_B - \nabla J_A^T H_B \nabla J_B \tag{3.38}$$

Following Nichol et al. (2018) we can rewrite this as:

$$= -(\nabla J_B^T H_A + \nabla J_A^T H_B) \nabla J_B \tag{3.39}$$
$$= -(\nabla_\theta (\nabla J_B^T \nabla J_A)) \nabla J_B \tag{3.40}$$

This last form is easy to compute with an automatic differentiation software and does not require explicitly computing the hessian. We also verify empirically that this quantity holds with commonly used small step-sizes.

The derivative of function interference can also be written similarly:

$$\bar{\rho}'_{AB} = \frac{\partial}{\partial\alpha} \nabla_{\theta'} f_A^T \nabla_{\theta'} f_B \tag{3.41}$$
$$= (\frac{\partial(\nabla_{\theta'} f_A)}{\partial\theta'} \frac{\partial\theta'}{\partial\alpha})^T \nabla_{\theta'} f_B + \nabla_{\theta'} f_A^T (\frac{\partial(\nabla_{\theta'} f_B)}{\partial\theta'} \frac{\partial\theta'}{\partial\alpha}) \tag{3.42}$$
$$= (-\nabla^2_{\theta'} f_A \nabla_\theta J(B))^T \nabla_{\theta'} f_B + \nabla_{\theta'} f_A^T (-\nabla^2_{\theta'} f_B \nabla_\theta J(B)) \tag{3.43}$$
$$\approx -\nabla J_B^T \bar{H}_A \nabla f_B - \nabla f_A^T \bar{H}_B \nabla J_B \tag{3.44}$$
$$= -(\nabla f_B^T \bar{H}_A + \nabla f_A^T \bar{H}_B) \nabla J_B \tag{3.45}$$

where by $\bar{H}$ we denote the Hessian of the function f itself rather than of its loss.

Note that for the parameterized function f_θ

$$\nabla_\theta J = \frac{\partial J}{\partial f} \frac{\partial f}{\partial\theta}$$

Let's write $\frac{\partial J}{\partial f} = \delta$. For any regression-like objective $(f-y)^2/2$, $\delta = (f-y)$. δ's sign will be positive if f needs to decrease, and negative if f needs to increase.

Let's rewrite the interference as:

$$\nabla_\theta J_\theta(A)^T \nabla_\theta J_\theta(B) = \delta_A \delta_B \nabla_\theta f_\theta(A)^T \nabla_\theta f_\theta(B)$$

Then notice that ρ' can be decomposed as follows. Let $g_{AB} = \nabla_\theta f_\theta(A)^T \nabla_\theta f_\theta(B)$, $g'_{AB} = \nabla_{\theta'} f_{\theta'}(A)^T \nabla_{\theta'} f_{\theta'}(B)$:

$$\begin{aligned}
\rho'_{reg;AB} =& \frac{\partial}{\partial\alpha}\delta_A\delta_B\nabla_{\theta'} f_A^T \nabla_{\theta'} f_B && (3.46)\\
=& \frac{\partial\delta_A}{\partial\alpha}\delta_B g'_{AB} + \frac{\partial\delta_B}{\partial\alpha}\delta_A g'_{AB} \\
&+ \delta_A\delta_B(\frac{\partial}{\partial\alpha}\nabla_{\theta'} f_A)^T\nabla_{\theta'} f_B + \delta_A\delta_B(\frac{\partial}{\partial\alpha}\nabla_{\theta'} f_B)^T\nabla_{\theta'} f_A && (3.47)\\
=& -\nabla_{\theta'} f_A^T \nabla_\theta J_B \delta_B g'_{AB} - \nabla_{\theta'} f_B^T \nabla_\theta J_B \delta_A g'_{AB} \\
&+ \delta_A\delta_B(-\bar{H}_{\theta';A}\nabla_\theta J_B)^T\nabla_{\theta'} f_B + \delta_A\delta_B(-\bar{H}_{\theta';B}\nabla_\theta J_B)^T\nabla_{\theta'} f_A && (3.48)
\end{aligned}$$

if we assume $\theta \approx \theta'$, $g \approx g'$ we can simplify

$$\begin{aligned}
&\approx - g\delta_B\delta_B g - 2\delta_B g\delta_A g_{BB} - \delta_A\delta_B\delta_B\nabla_\theta f_B \bar{H}_A \nabla_\theta f_B - \delta_A\delta_B\delta_B\nabla_\theta f_B \bar{H}_B \nabla_\theta f_A && (3.49)\\
&= - g_{AB}^2\delta_B^2 - 2\delta_A\delta_B g_{AB} g_{BB} - \delta_A\delta_B^2\nabla_\theta f_B(\bar{H}_A\nabla f_B + \bar{H}_B\nabla_\theta f_A) && (3.50)
\end{aligned}$$

We can also compute ρ' for TD(0) assuming that the target is not frozen and is influenced by an update to θ. Again we want $\partial/\partial\alpha[g'_{AB}]$ for an update at B, interference at A, assuming that B' is a successor state of B used for the TD update, and A' a successor of A in δ_A:

$$\begin{aligned}
\theta' &= \theta - \alpha\delta_B\nabla_\theta f_B && (3.51)\\
&= \theta - \alpha(f_B - (r + \gamma f_{B'})\nabla_\theta f_B && (3.52)
\end{aligned}$$

Also note that:

$$\begin{aligned}
\frac{\partial\delta_A}{\partial\alpha} &= \left(\frac{\partial f_A}{\partial\theta'}\frac{\partial\theta'}{\partial\alpha} - \gamma\frac{\partial f_{A'}}{\partial\theta'}\frac{\partial\theta'}{\partial\alpha}\right) && (3.53)\\
&= -\delta_B(\nabla_{\theta'} f_A{}^T\nabla_\theta f_B - \gamma\nabla_{\theta'} f_{A'}{}^T\nabla_\theta f_B) && (3.54)
\end{aligned}$$

Let $g_{AB} = \nabla_\theta f_A^T\nabla_\theta f_B$, $g'_{AB} = \nabla_{\theta'} f_A^T\nabla_{\theta'} f_B$ and $g^{\backslash}_{AB} = \nabla_{\theta'} f_A^T\nabla_\theta f_B$:

$$\begin{aligned}
\rho_{TD;AB} &= \frac{\partial}{\partial\alpha}\left[\delta_A\delta_B\nabla_{\theta'} f_A^T\nabla_{\theta'} f_B\right] && (3.55)\\
&= \frac{\partial}{\partial\alpha}\delta_A\delta_B\nabla_{\theta'} f_A^T\nabla_{\theta'} f_B \\
&\quad + \delta_A\frac{\partial}{\partial\alpha}\delta_B\nabla_{\theta'} f_A^T\nabla_{\theta'} f_B \\
&\quad + \delta_A\delta_B\frac{\partial}{\partial\alpha}\nabla_{\theta'} f_A^T\nabla_{\theta'} f_B \\
&\quad + \delta_A\delta_B\nabla_{\theta'} f_A^T\frac{\partial}{\partial\alpha}\nabla_{\theta'} f_B && (3.56)\\
&= -\delta_B(g^{\backslash}_{AB} - \gamma g^{\backslash}_{A'B})\delta_B g'_{AB} \\
&\quad - \delta_B(g^{\backslash}_{BB} - \gamma g^{\backslash}_{B'B})\delta_A g'_{AB} \\
&\quad + \delta_A\delta_B(-\bar{H}_{\theta';A}\nabla_\theta J_B)^T\nabla_{\theta'} f_B + \delta_A\delta_B(-\bar{H}_{\theta';B}\nabla_\theta J_B)^T\nabla_{\theta'} f_A && (3.57)
\end{aligned}$$

which again if we assume $\theta' \approx \theta, g_{AB} \approx g'_{AB} \approx g^{\setminus}_{AB}$, we can simplify to:

$$\rho'_{TD;AB} = -\delta_B^2 g_{AB}(g_{AB} - \gamma g_{A'B}) - \delta_A\delta_B g_{AB}(g_{BB} - \gamma g_{B'B}) \\ -\delta_A\delta_B^2 \nabla_\theta f_B(\bar{H}_A \nabla_\theta f_B + \bar{H}_B \nabla_\theta f_A) \quad (3.58)$$

3.2.4.2 Architectures, hyperparameter ranges, and other experimental details

We use the PyTorch library (Paszke et al., 2019a) for all experiments. To efficiently compute gradients for a large quantity of examples at a time we use the backpack library (Dangel et al., 2020).

To summarize the choice of problems, we run natural images experiments first to get a more accurate comparison of the generalization gap between RL and SL. We then run Atari experiments to analyze information propagation, TD(λ), and the local coherence of targets, because Atari agents (1) have long term decision making which highlights the issues of using TD for long term reward predictions (which is TD's purpose) and (2) are a standard benchmark.

Figure 3.4, 3.5, 3.6 and 3.16 In order to generate these figures we train classifiers, regression models, DDQN agents and REINFORCE agents.

Models trained on SVHN and CIFAR10, either for SL, DDQN, or REINFORCE, use a convolutional architecture. Let n_h be the number of hiddens and n_L the number of extra layers. The layers are:

- Convolution, 3 in, n_h out, filter size 5, stride 2
- Convolution, n_h in, $2n_h$ out, filter size 3
- Convolution, $2n_h$ in, $4n_h$ out, filter size 3
- n_L layers of Convolution, $4n_h$ in, $4n_h$ out, filter size 3, padding 1
- Linear, $4n_h \times 10 \times 10$ in, $4n_h$ out
- Linear, $4n_h$ in, n_o out.

All layers except the last use a Leaky ReLU (Maas et al., 2013) activation with slope 0.01 (note that we ran a few experiments with ReLU and tanh activations out of curiosity, except for the slightly worse training performance the interference dynamics remained fairly similar). For classifiers n_o is 10, the number of classes. For agents n_o is 10+4, since there are 10 classes and 4 movement actions.

Models trained on the California Housing dataset have 4 fully-connected layers: 8 inputs, 3 Leaky ReLU hidden layers with n_h hiddens, and a linear output layer with a single output.

Models trained on the SARCOS dataset have $2{+}n_L$ fully-connected layers: 21 inputs, $1{+}n_L$ Leaky ReLU hidden layers with n_h hiddens, and a linear output layer with 8 outputs.

Let n_T be the number of training seeds. We use the following hyperparameter settings:

- SVHN, $n_h \in \{8, 16, 32\}$, $n_L \in \{0, 1, 2, 3\}$,
  $n_T \in \{20, 100, 250, 500, 1000, 5000, 10000, 50000\}$
- CIFAR10, $n_h \in \{16, 32, 64\}$, $n_L \in \{0, 1, 2, 3\}$,
  $n_T \in \{20, 100, 250, 500, 1000, 5000, 10000, 50000\}$
- SARCOS, $n_h \in \{16, 32, 64, 128, 256\}$, $n_L \in \{0, 1, 2, 3\}$,
  $n_T \in \{20, 100, 250, 500, 1000, 5000, 10000, 44484\}$
- California Housing, $n_h \in \{16, 32, 64, 128\}$,
  $n_T \in \{20, 100, 250, 500, 1000, 5000, 10000\}$

For SVHN and CIFAR10, we use the same architecture and hyperparameter ranges for classification, DDQN and REINFORCE experiments. Each hyperparameter setting is run with 3 or more seeds. The seeds affect the initial parameters, the sampling of minibatches, and the sampling of ϵ-greedy actions.

Note that while we run REINFORCE on SVHN and CIFAR, we do not spend a lot of time analyzing its results, due the relatively low relevance of PG methods to the current work. Indeed, the goal was only to highlight the difference in trends between TD and PG, which do indicate that the two have different behaviours. Policy gradient methods do sometimes rely on the TD mechanism (e.g. in Actor-Critic), but they use different update mechanisms and deserve their own independent analysis, see for example Ilyas et al. (2018).

For optimizers, we use the standard settings of PyTorch:

- Adam, $\beta = (0.9, 0.999)$, $\epsilon = 10^{-8}$
- RMSProp, $\alpha = 0.99$, $\epsilon = 10^{-8}$
- Momentum SGD, $\beta = 0.9$ (with Nesterov momentum off)

Figure 3.7, 3.8, 3.9, 3.10, and 3.14 Figure 3.7 is obtained by training models for 500k steps with a standard DQN architecture (Mnih et al., 2013): 3 convolutional layers with kernels of shape $4 \times 32 \times 8 \times 8$, $32 \times 64 \times 4 \times 4$, and $64 \times 64 \times 3 \times 3$ and with stride 4, 2, and 1 respectively, followed by two fully-connected layers of shape 9216×512 and $512 \times |\mathcal{A}|$, $\mathcal{A}$ being the legal action set for a given game. All activation are leaky ReLUs except for the last layer which is linear (as it outputs value functions). Experiments are run on MsPacman, Asterix and Seaquest for 10 runs each. A learning rate of 10^{-4} is used, with L2 weight regularization of 10^{-4}. We use $\gamma = 0.99$, a minibatch size of 32, an ϵ of 5% to generate $\mathcal{D}^*$, and a buffer size of 500k. The random seeds affect the generation of $\mathcal{D}^*$, the weight initialization, the minibatch sampling, and the choice of actions in ϵ-greedy rollouts.

As per the previous figure, for Figure 3.8 we run experiments with a standard DQN architecture, train our policy evaluation models for 500k and our control models for 10M steps. When boostrapping to a frozen network, the frozen network is updated every 10k updates.

Figures 3.9, 3.10, and 3.14 also use results from these experiments.

Figure 3.12, 3.13, and 3.14 The experiments of Figure 3.12 are run for 500k steps, as previously described, on MsPacman. λ-targets are computed with the forward view, using the frozen network to compute the target values – this allows us to cheaply recompute all λ-targets once every 10k steps when we update the frozen network. Each setting is run with 5 random seeds.

Figures 3.13 and 3.14 also use results from these experiments.

Figure 3.15 Figure 3.15 reuses the results of Figure 3.8's policy evaluation experiments run with Adam.

Figure 3.17 Figure 3.17 uses the same experiment setup as in the Atari regression experiments on MsPacman, as well as policy evaluation experiments on MsPacman as previously described, all the while measuring individual terms of ρ'_{reg} and $\rho'_T D$. Experiments are only run for the first 100k steps. Minibatches of size 32 are used.

3.3 Discussion

In this chapter we've seen that in deep neural networks, memorization and generalization are not necessarily at odds. In supervised learning, with structured data, DNNs learn to rely on a few samples and in a sense memorize them, while at the same time learning simple patterns, applicable to a wide range of inputs. Both these mechanisms enable generalization, the ability to make correct predictions on novel inputs.

We've then taken this investigation to deep Reinforcement Learning, where we found that some behaviours related to generalization, in particular interference, are heavily altered in comparison to supervised learning. These suggests that there is a lack of generalization even *within* the training data–or perhaps more precisely, a lack of parameter sharing between samples of the training data. What should we make of it?

Indeed, RL is generally considered a harder problem than supervised learning due to the non-i.i.d. nature of the data. Hence, the fact that TD-style methods require more samples than supervised learning when used with neural networks is not necessarily surprising. However, with the same data and the same final targets (the "true" value function), it is

not clear why TD updates lead to parameters that generalize worse than in supervised learning. Indeed, our results show that the interference of a converged model evolves differently as a function of data and capacity in TD than in supervised learning.

Our results also show that Q-Learning generalizes poorly, leading to DNNs that memorize the training data (not unlike table lookup). Our results also suggest that TD(λ), although not widely used in recent DRL, improves generalization. Additionally, we found differences between Adam and RMSProp that we initially did not anticipate. Very little work has been done to understand and improve the coupling between optimizers and TD, and our results indicate that this is an important future work direction. Indeed, a method aware of this coupling is later proposed in §4.1 (Bengio et al., 2020b).

While a full description of the mechanisms that cause TD methods to have such problems remains elusive, we find that understanding the evolution of gradient interference reveals intriguing differences in memorization behaviours between the supervised and temporal difference objectives, and hint at the importance of stable targets in bootstrapping.

Looking forward, this work suggests that the RL community should pay special attention to the current research on generalization in DNNs, as naively approaching the TD bootstrapping mechanism as a supervised learning problem does not seem to leverage the full generalization potential of DNNs.

This is especially visible in what has been driving the recent impressive successes of deep RL (Silver et al., 2017; Hessel et al., 2018; Vinyals et al., 2019), relatively simple methods, massively scaled. It seems probable that the need for these methods to leverage many lifetimes of data results from poor generalization mechanisms in deep RL.

Another problem this work alludes to is the need to properly define generalization in RL. There are likely to be multiple axes to it, state-generalization, task-generalization, action-generalization (Cobbe et al., 2019; Packer et al., 2018; Zhang et al., 2018a; Justesen et al., 2018; Witty et al., 2018; Igl et al., 2020; Lan et al., 2022); although these superficially appear to be separate issues, recent empirical results all suggest they are in fact deeply intertwined.

4

Optimization and Temporal Difference

This chapter synthesizes my contribution on a novel optimizer for temporal difference learning methods (Bengio et al., 2020b).

A central part of Machine Learning is, of course, the *learning* aspect, which happens in a myriad of different ways. In deep learning, most of it happens through **gradient descent** and its many variants taking roots in stochastic optimization (Bottou, 1998).

This proliferation of **optimization** tools has led to a number of advances in supervised learning, which we will start the chapter by discussing. Surprisingly, deep reinforcement learning methods have been able to use these tools *as is* and obtain great empirical results, despite the technical challenges that arise from non-stationarity in RL. In the rest of the chapter, we will discuss a novel optimization method that explicitly assumes this non-stationarity, and uses it to enhance the performance of optimization process of temporal difference learning methods. We find that by trying to correct for *staleness*, one can learn the value function much faster.

In the next sections, we again closely follow the original material of my contribution (Bengio et al., 2020b). The code for the methods described here is available at https://github.com/bengioe/staleness-corrected-momentum.

4.1 Accelerated Methods, Bootstrapping and Staleness

4.1.1 On Supervised Learning Tools in Reinforcement Learning

Typical stochastic convex optimization problems are formulated as finding parameters θ^* for which the expected value $J(\theta) = \mathbb{E}_X[J(X;\theta)]$ is minimized for some convex ($\nabla^2_\theta J > 0$) function $J(\theta)$.

Since the expectation is generally intractable, we instead rely of stochastic samples of X to update the current estimate of θ using a *step-size* a_t as:

$$\theta_{t+1} = \theta_t - a_t \nabla_{\theta_t} J(X_t;\theta_t). \tag{4.1}$$

This follows Robbins and Monro (1951)'s method, where we estimate θ for which $\nabla_\theta J(\theta) = 0$. This method converges if the following assumptions hold:

- we can sample the random variable $G(\theta)$ such that $\mathbb{E}_X[G] = \nabla_\theta J(\theta)$,
- $G(\theta)$ is bounded
- the step size a_t is such that $\sum_t a_t = \infty$ and $\sum_t a_t^2 < \infty$

Note that the samples of G here are formed through another random variable X, which represents the data. For these conditions to hold, sampling of X must follow the same distribution throughout. In other words, $X_1, X_2, \ldots$ are **identically and independently distributed** (i.i.d.). This method is commonly referred to as **stochastic gradient descent** (SGD).

Supervised Deep Learning breaks a central assumption here, that J is convex. Deep Reinforcement Learning, which is broadly based on the same update mechanism, *also* breaks the i.i.d. assumption. Although there are results on convergence in non-convex non-i.i.d. settings, they are not always readily applicable to DNNs.

There is a broad literature on why DNNs still converge, even though their optimization landscape is non-convex, but it is not the focus of this work. Let's nonetheless highlight a key result discovered by Dauphin et al. (2014); one well known danger of non-convex optimization is that of local minima (Gori and Tesi, 1992), whereby the iterative process above converges to a locally convex region of $J(\theta)$ for which $J(\theta) > J(\theta^*)$. Dauphin et al. (2014) find that this turns out not to be a problem for DNNs, due to the high-dimensionality of θ, one is much more likely to encounter a saddle point (where $\nabla_\theta J(\theta) = 0$) than a local minima (and there are many methods to escape saddle points). Indeed, as Dauphin et al. (2014) point out, there are exponentially more saddle points in an N variable Gaussian process than local minima as N increases.

By relaxing the i.i.d. assumption on the data distribution, we enter a hazardous territory where there are no guarantees, and many pitfalls (van Hasselt et al., 2018). We can in particular highlight two categories of so-called **non-stationarity**, whereby $X_1, X_2, \ldots$ are no longer independent, nor identically distributed. As an agent learns and collects experience, the data it sees changes. More fundamentally, a host of RL methods rely on an agent's own predictions of the future to learn, i.e. so-called **bootstrapping**. This means that as an agent learns, its own learning signal changes, which is another significant source of non-stationarity.

Both these non-stationarites break the "identically distributed" assumption, but more importantly they break the "independently distributed" assumption, as what an agent learns now depends on what it has learned in the past.

These fundamental questions highlight the necessity to understand optimization in Deep RL much better, and to formulate methods adapted to this setting.

4.1.2 Accelerated Methods and Staleness

While SGD is a fundamental and well understood method, it can be rather slow to converge. In a sense, because of the stochastic nature of each iterate, all the information used previously is lost.

A notable method to make SGD faster is **momentum**, a so-called **accelerated method**, which combines these past iterates in a velocity term to obtain a stronger sense of direction. An interesting 3-dimensional analogy is to imagine SGD as someone walking downhill, adjusting their direction after every step to find the steepest descent; momentum would be a large heavy ball, rolling downhill via gravity and its own *momentum*.

The usual form of momentum (Polyak, 1964; Sutskever et al., 2013) in SGD maintains an exponential moving average with factor β of gradients w.r.t. to some objective J, changing parameters $\theta_t \in \mathbb{R}^n$ with learning rate α:

$$\mu_t = \beta\mu_{t-1} + (1-\beta)\nabla_{\theta_{t-1}} J_t(\theta_{t-1}) \tag{4.2}$$

$$\theta_t = \theta_{t-1} - \alpha\mu_t \tag{4.3}$$

We write J_t to fold in the dependency on X_t. Note that other similar forms of this update exist, notably Nesterov's accelerated gradient method (Nesterov, 1983), as well as undampened methods that omit $(1-\beta)$ in (4.2) or replace $(1-\beta)$ with α, found in popular deep learning packages (Paszke et al., 2019b).

We make the observation that, at time t, the gradients accumulated in μ are *stale*. They were computed using past parameters rather than θ_t, and in general we'd expect $\nabla_{\theta_t} J_t(\theta_t) \neq \nabla_{\theta_k} J_t(\theta_k)$, $k < t$. As such, the update in (4.3) is a *biased* update.

Note that here we do not use "bias" as it relates to convergence (indeed it appears that momentum SGD is well behaved in that regard, see Wang et al., 2021). Rather, we note that the update differs in expectation (in the limit of infinite an batch size used to compute $\nabla_\theta J(\theta)$) from a gradient descent update, because it depends on the previous location of the parameters in the optimization process. In that sense, we say that this dependency on previous gradient updates induces some staleness, some bias, to the update.

In supervised learning where one learns a mapping from x to y, this staleness only has one source: θ changes but the target y stays constant. As we've discussed in the last section, bootstrapping introduces an additional non-stationary, and we argue that in TD

learning, momentum becomes **doubly** stale: not only does the value network change, but the target (the equivalent of y) itself changes[1] with every parameter update.

Let's recall the TD(0) update:

$$\theta_{t+1} = \theta_t - \alpha \nabla_{\theta_t} \left(V_{\theta_t}(S_t) - (R(S_t) + \gamma \bar{V}_{\theta_t}(S_{t+1})) \right)^2, \tag{4.4}$$

with $\bar{V}$ meaning we consider V constant for the purpose of gradient computation.

When θ changes, not only does $V(s)$ change, but $V(s')$ as well. The update itself changes, making past gradients stale and less aligned with recent gradients (even more so when there is gradient interference (Liu et al., 2019a; Achiam et al., 2019; Bengio et al., 2020a), constructive *or* destructive, as we have seen in the previous chapter).

Note that several sources of bias already exist in TD learning, notably the traditional parametric bias (of the bias-variance tradeoff when selecting capacity, see Neal et al., 2019), as well as the bootstrapping bias (of the error in $V(s')$ when using it as a target; using a frozen target prevents this bias from compounding, see van Hasselt et al., 2018). We argue that the staleness in momentum we describe is an additional form of bias, slowing down or preventing convergence. This has been hinted at before, e.g. Gupta (2020) suggests that momentum hinders learning in linear TD(0).

Part of the success of DNNs, including when applied to TD learning, is the use of adaptive or accelerated optimization methods (Hinton et al., 2012; Sutskever et al., 2013; Kingma and Ba, 2015) to find good parameters. In this chapter we investigate and extend the momentum algorithm (Polyak, 1964) as applied to TD learning in DNNs. While accelerated TD methods have received some attention in the literature, this is typically done in the context of linear function approximators (Baxter and Bartlett, 2001; Meyer et al., 2014; Pan et al., 2017; Gupta et al., 2019; Gupta, 2020; Sun et al., 2021), and while some studies have considered the mix of DNNs and TD (Zhang et al., 2019; Romoff et al., 2021), many are limited to a high-level analysis of hyperparameter choices for *existing* optimization methods (Sarigül and Avci, 2018; Andrychowicz et al., 2020); or indeed the latter are simply applied as-is to train RL agents (Mnih et al., 2013; Hessel et al., 2018).

As a first step in going beyond the naive use of supervised learning tools in RL, we examine momentum. We argue that momentum, especially as it is used in conjunction with TD and DNNs, adds an additional form of bias which can be understood as the staleness of accumulated information. We quantify this bias, and propose a corrected momentum algorithm that reduces this staleness and is capable of improving performance.

[1]Interestingly, even in most recent value-based control works (Hessel et al., 2018) a (usually *frozen*) copy is used for *stability*, meaning that the target only changes when the copy is updated. This is considered a "trick" which it would be compelling to get rid of, since it slows down learning, and since most recent policy-gradient methods (which still use a value function) do not make use of such copies (Schulman et al., 2017).

4.1.3 Related Work

To the best of our knowledge, no prior work attempts to derive a corrected momentum-SGD update adapted to the Temporal Difference method. That being said, a wealth of papers are looking to accelerate TD and related methods.

On momentum, traces, and gradient acceleration in TD

From an RL perspective, our work has some similarity to the so-called eligibility traces mechanism. In particular, in the True Online TD(λ) method of van Seijen and Sutton (2014), the authors derive a *strict-online* update (i.e. weights are updated at every MDP step, using only information from past steps, rather than future information as in the λ-return perspective) where the main mechanism of the derivation lies in finding an update by assuming (at least analytically) that one can "start over" and reuse all past data iteratively at each step of training, and then from this analytical assumption derive a recursive update (that doesn't require iterating through all past data). The extra values that have to be kept to compute the recursive updates are then called traces. This is akin to how we will later conceptualize the "ideal" momentum μ^*, (4.6), and derive $\hat{\mu}$.

The conceptual similarities of the work of van Seijen and Sutton (2015) with our work are also interesting. There, the authors analyse what "retraining from scratch" means (i.e., again, iteratively restarting from $\theta_0 \in \mathbb{R}^m$) but with some ideal target θ^* (e.g. the current parameters) by redoing sequentially all the TD(0) updates using θ^* for all the n transitions in a replay buffer, costing $O(nm)$. They derive an online update showing that one can continually learn at a cost of $O(m^2)$ rather than paying $O(nm)$ at each step. The proposed update is also reminiscent of our method in that it aims to perform an approximate batch update without computing the entire batch gradient, and also maintains extra momentum-like vectors and matrices. We note that the derivation there only works in the linear TD case.

In a way, such an insight can be found in the original presentation of TD(λ) of Sutton (1988), where the TD(λ) parameter update is written as (equation (4) in the original paper, but with adapted notation):

$$\Delta\theta_t = \alpha[r_t + \gamma V_{\theta_t}(s_{t+1}) - V_{\theta_t}(s_t)] \sum_{k=1}^{t} \lambda^{t-k} \nabla_{\theta_t} V_{\theta_t}(s_k)$$

Remark the use of θ_t in the sum; in the linear case since $\nabla_{\theta_t} V_{\theta_t}(s_k) = \phi(s_k)$, the sum does not depend on θ_t and thus can be computed recursively. A posteriori, if one can find a way to cheaply compute $\nabla_{\theta_t} V_{\theta_t}(s_k)\ \forall k$, perhaps using the method we propose, it may be an interesting way to perform TD(λ) using a non-linear function approximator.

Our analysis is also conceptually related to the work of Schapire and Warmuth (1996), where a worst-case analysis of TD*(λ) is performed using a *best-case learner* as the per-

formance upper bound. This is similar to our *momentum oracle*; just as the momentum oracle is the "optimal" approximation of the accumulation gradients coming from all past training examples, the best-case learner of Schapire and Warmuth (1996) is the set parameters that is optimal when one is allowed to look at all past training examples (in contrast to an online TD learner).

Before moving on from TD(λ), let us remark that eligibility traces and momentum, while similar, estimate different quantities. The usual (non-replacing) traces estimate the exponential moving average of the gradient of V_θ, while momentum does so for the objective J (itself a function of V_θ):

$$\mathbf{e}_t = (1-\lambda)\sum_k^t \lambda^{t-k}\nabla_\theta V_\theta, \quad \mu_t = (1-\beta)\sum_k^t \beta^{t-k}\nabla_\theta J(V_\theta)$$

Our method also has similarities with residual gradient methods (Baird, 1995). A recent example of this is the work of Zhang et al. (2019), who adapt the residual gradient for deep neural networks. Residual methods learn by taking the gradient of the TD loss with respect to both the current value and the next state value $V(S')$, but this comes at the cost of requiring two independent samples of S' (except in deterministic environments).

Similarly, our work is related to the "Gradient TD" family of methods (Sutton et al., 2008, 2009b). These methods attempt to maintain an expectation (over states) of the TD update, which allows to directly optimize the Bellman objective. While the exact relationship between GTD and "momentum TD" is not known, they both attempt to maintain an "expected update" and adjust parameters according to it; the first approximates the one-step linear TD solution, while the latter approximates the one-step batch TD update. Note that linear GTD methods can also be accelerated with momentum-style updates (Meyer et al., 2014), low-rank approximations for part of the Hessian (Pan et al., 2017), and adaptive learning rates (Gupta et al., 2019).

Interestingly, while GTD is a "proper" gradient method, TD as it is generally used is not, although the latter is widely used because it does not suffer from the double sampling problem. Such an observation is part of the motivation of trying to address issues in the use of momentum applied to TD methods.

More directly related to this work is that of Sun et al. (2021), who show convergence properties of a rescaled momentum for linear TD(0). While most (if not every) deep reinforcement learning method implicitly uses some form of momentum and/or adaptive learning rate as part of the deep learning toolkit, Sun et al. (2021) properly analyse the use of momentum in a (linear) TD context. Gupta (2020) also analyses momentum in the context of a linear TD(0) and TD(λ), with surprising negative results suggesting naively applying momentum may hurt stability and convergence in minimal MDPs.

Another TD-aware adaptive method is that of Romoff et al. (2021), who derive per-parameter adaptive learning rates, reminiscent of RMSProp (Hinton et al., 2012), by considering a (diagonal) Jacobi preconditioning that takes into account the bootstrap term in TD.

Finally, we note that, as far as we know, recent deep RL works all use some form of adaptive gradient method, Adam (Kingma and Ba, 2015) being an optimizer of choice, closely followed by RMSProp (Hinton et al., 2012); notable examples of such works include those of Mnih et al. (2013), Schulman et al. (2017), Hessel et al. (2018), and Kapturowski et al. (2019). We also note the work of Sarigül and Avci (2018), comparing various SGD variants on the game of Othello, showing significant differences based on the choice of optimizer.

On Taylor approximations

Balduzzi et al. (2017) note that while theory suggests that Taylor expansions around parameters should not be useful because of the "non-convexity" of ReLU neural networks, there nonetheless exists local regions in parameter space where the Taylor expansion is consistent. Much earlier work by Engelbrecht (2000) also suggests that Taylor expansions of small sigmoid neural networks are easier to optimize. Using Taylor approximations around parameters to find how to prune neural networks also appears to be an effective approach with a long history (LeCun et al., 1990; Hassibi and Stork, 1993; Engelbrecht, 2001; Molchanov et al., 2016).

On policy-gradient methods and others

While not discussed in this chapter, another class of methods used to solve RL problems are PG methods. They consist in taking gradients of the objective w.r.t. a directly parameterized policy (rather than inducing policies from value functions). We note in particular the work of Baxter and Bartlett (2001), who analyse the bias of momentum-like cumulated policy gradients (referred to as traces therein), showing that β the momentum parameter should be chosen such that $1/(1-\beta)$ exceeds the mixing time of the MDP.

Let us also note the method of Vieillard et al. (2020), Momentum Value Iteration, which uses the concept of an exponential moving average objective for a decoupled (with its own parameters) action-value function from which the greedy policy being evaluated is induced. This moving average is therein referred to as *momentum*; even though it is not properly speaking the optimizational acceleration of Polyak (1964), its form is similar.

4.2 Correcting Staleness in Momentum

4.2.1 Identifying Bias

Here we propose an experimental protocol to quantify the previously described bias of momentum. We assume we are minimizing some objective in the minibatch or online setting. In such a stochastic setting, momentum is usually understood as a variance reduction method, or as approximating the large-batch setting (Botev et al., 2017), but as discussed above, momentum also induces bias in the optimization.

We note that μ_t, the momentum at time t, can be rewritten as:

$$\mu_t = (1-\beta)\sum_{i=1}^{t}\beta^{t-i}\nabla_{\theta_i} J_i(\theta_i), \tag{4.5}$$

and argue that an ideal *unbiased* momentum μ_t^* would approximate the large batch case by only discounting past minibatches and using current parameters rather than past parameters:

$$\mu_t^* \stackrel{\text{def}}{=} (1-\beta)\sum_{i=1}^{t}\beta^{t-i}\nabla_{\theta_t} J_i(\theta_t). \tag{4.6}$$

Note that the only difference between (4.5) and (4.6) is the use of θ_i vs θ_t. The only way to *exactly* compute μ_t^* is to recompute the entire sum after every parameter update. We will consider this our **unbiased oracle**. To compute μ_t^* empirically, since beyond a certain power β^k becomes small, we will use an effective horizon of $h = 2/(1-\beta)$ steps (i.e. start at $i = t-h$ rather than at $i = 1$).

We call the difference between μ_t and μ_t^* the *bias*, but note that in the overparameterized stochastic gradient case, minor differences in gradients can quickly send parameters in different regions. This makes the direct measure of $\|\mu_t - \mu_t^*\|$ uninformative. Instead, we measure the **optimization bias** by simply comparing the loss of a model trained with μ_t against that of a model trained with μ_t^*.

Finally, we note that, in RL, momentum approximating the batch case is related to (approximately) replaying an entire buffer at once (instead of sampling transitions). This has been shown to also have interesting forms in the linear case (van Seijen and Sutton, 2015), reminiscent of the correction derived below. We also note that, while the mathematical expression of momentum and eligibility traces (Sutton, 1988) *look* fairly similar, they estimate a very different quantity (as we have seen in §4.1.3).

4.2.2 Correcting Bias

Here we propose a way to approximate μ^*, and so derive an approximate correction to the bias in momentum for supervised learning, as well as for Temporal Difference (TD) learning.

We consider here the simple regression and TD(0) cases, for the online (minibatch size 1) case. We show the full derivations, and results for any minibatch size, TD(λ) and n-step TD, in §4.2.3. In least squares regression with loss δ^2 we can write the gradient g_t as:

$$g_t(\theta_t) = (y_t - f_{\theta_t}(x_t))\nabla_{\theta_t} f_{\theta_t}(x_t) = \nabla_{\theta_t}\delta_t^2/2. \tag{4.7}$$

We would like to "correct" this gradient as we move away from θ_t. A simple way to do so is to compute the Taylor expansion around θ_t of g_t:

$$\begin{aligned} g_t(\theta_t + \Delta\theta) &= g_t(\theta_t) + \nabla_{\theta_t} g_t(\theta_t)^\top \Delta\theta + o(\|\Delta\theta\|_2^2), & (4.8)\\ &\approx g_t(\theta_t) + (\delta\nabla^2_{\theta_t} f_{\theta_t}(x_t) - \nabla_{\theta_t} f_{\theta_t}(x_t) \otimes \nabla_{\theta_t} f_{\theta_t}(x_t))^\top \Delta\theta, & (4.9)\end{aligned}$$

where $\otimes$ is the outer product, ∇^2 the second derivative. We note that the term multiplying $\Delta\theta$ is commonly known as "the Hessian" of the loss (Bottou, 1998). We also note that Taylor expansions around parameters have a rich history in deep learning (LeCun et al., 1990; Molchanov et al., 2016), and that in spite of its non-linearity, the parameter space of a deep network is filled with locally consistent regions (Balduzzi et al., 2017) in which Taylor expansions are accurate.

The same correction computation can be made for TD(0) with TD loss δ^2. Remember that we write t as the learning time; we denote an MDP transition as (s_t, a_t, r_t, s'_t), making no assumption on the distribution of transitions used for learning:

$$\begin{aligned} g_t(\theta_t) &= (V_{\theta_t}(s_t) - r_t - \gamma V_{\theta_t}(s'_t))\nabla_{\theta_t} V_{\theta_t}(s_t) = \nabla_{\theta_t}\delta_t^2/2, & (4.10)\\ g_t(\theta_t + \Delta\theta) &\approx g_t(\theta_t) + (\nabla_{\theta_t}(V_{\theta_t}(s_t) - \gamma V_{\theta_t}(s'_t)) \otimes \nabla_{\theta_t} V_{\theta_t}(s_t) + \delta\nabla^2_{\theta_t} V_{\theta_t}(s_t))^\top \Delta\theta. & (4.11)\end{aligned}$$

Here, because we are using a semi-gradient method, the term multiplying $\Delta\theta$ is not exactly the Hessian: when computing $\nabla_\theta \delta^2$, we hold $V(s')$ constant, but when computing $\nabla_\theta g$, we need to consider $V_\theta(s')$ a function of θ as it affects g, and so compute its gradient.[2]

Without loss of generality, let us write equations like (4.9) and (4.11) using the matrix $Z_t \in \mathbb{R}^{n\times n}$:

$$g_t(\theta_t + \Delta\theta) \approx g_t(\theta_t) + Z_t^\top \Delta\theta, \tag{4.12}$$

[2]This computation of the gradient of $V_\theta(s')$ may remind the reader of the so-called *full gradient* or *residual gradient* (Baird, 1995), but its purpose here is very different: we care about learning using semi-gradients, TD(0) is a principled algorithm, but we also care about *how* this semi-gradient evolves as parameters change, and thus we need to compute $\nabla_\theta V_\theta(s')$.

where the form of Z_t, which we will refer to as the "Taylor term", depends on the loss (e.g. the Hessian in (4.9)).

Recall that in (4.6) we define μ^* as the discounted sum of gradients using θ_t for losses J_i, $i \leq t$. We call those gradients *corrected* gradients, $g_i^t := g_i(\theta_i + (\theta_t - \theta_i))$. At timestep t we update θ_t with $\alpha\mu_t$, thus we substitute $\Delta\theta = \theta_t - \theta_i = -\alpha \sum_{k=i}^{t} \mu_k$ in (4.12) and get:

$$\hat{g}_i^t \stackrel{\text{def}}{=} g_i(\theta_i) - \alpha Z_i^\top \sum_{k=i}^{t-1} \mu_k \approx g_i^t. \tag{4.13}$$

We can now approximate the unbiased momentum μ_t^* using (4.13), which we denote $\hat{\mu}$:

$$\hat{\mu}_t \stackrel{\text{def}}{=} (1-\beta) \sum_{i=1}^{t} \beta^{t-i} \hat{g}_i^t \tag{4.14}$$

$$= \mu_t - \alpha(1-\beta) \sum_{k=1}^{t-1} \sum_{i=1}^{k} \beta^{t-i} Z_i^\top \hat{\mu}_k. \tag{4.15}$$

Noting that μ_t can be computed as usual and that the second term of (4.15) has a recursive form, which we denote η_t (see §4.2.3), we rewrite $\hat{\mu}$ as follows:

$$\hat{\mu}_t = \mu_t - \eta_t \tag{4.16}$$

$$\eta_t = \beta\eta_{t-1} + \alpha\beta\zeta_{t-1}^\top \hat{\mu}_{t-1} \qquad \text{(correction term)} \tag{4.17}$$

$$\mu_t = (1-\beta) \sum_{i=1}^{t} \beta^{t-i} g_i = (1-\beta) g_t + \beta\mu_{t-1} \qquad \text{(normal momentum)} \tag{4.18}$$

$$\zeta_t = (1-\beta) \sum_{i=1}^{t} \beta^{t-i} Z_i = (1-\beta) Z_t + \beta\zeta_{t-1}. \qquad \text{(“momentum” of Taylor terms)} \tag{4.19}$$

Algorithmically, this requires maintaining 2 vectors μ and η of size n, the number of parameters, and one matrix ζ of size $n \times n$. In practice, we find that only maintaining the diagonal (or block-diagonal) of ζ can also work and can avoid the quadratic growth in n.

The computation of Z also calls on computing second order derivatives $\nabla^2 f$ (e.g. in (4.9) and (4.11)), which is impractical for large architectures. In this work, as is commonly done due to the usually small magnitude of $\nabla^2 f$ (Bishop, 2006b, section 5.4), we ignore them and only rely on the outer product of gradients.

Ignoring the second derivatives, computing Z for TD(0) requires 3 backward passes, for $g = \nabla\delta^2$, $\nabla\gamma V(s')$, and $\nabla V(s)$. In the online case $g = \delta\nabla V(s)$, requiring only 2 backward passes, but in the more general minibatch of size $m > 2$ case, it is more efficient with modern automatic differentiation packages to do 3 backward passes than $2m$ passes (see §4.2.3.2).

Finally, we note that this method easily extends to forward-view TD(λ) and n-step TD methods, all that is needed is to compute Z appropriately (see §4.2.3.1).

4.2.3 Derivation of the momentum correction

Let us recall some definitions. In momentum, μ_t can be rewritten as:

$$\mu_t = (1-\beta)\sum_{i=1}^{t} \beta^{t-i} \nabla_{\theta_i} J_i(\theta_i) \tag{4.20}$$

We argue that an ideal *unbiased* momentum μ_t^* would approximate the large batch case by only discounting past minibatches and using current parameters θ_t rather than past parameters θ_i:

$$\mu_t^* \stackrel{\text{def}}{=} (1-\beta)\sum_{i=1}^{t} \beta^{t-i} \nabla_{\theta_t} J_i(\theta_t) \tag{4.21}$$

Note that the only difference between (4.20) and (4.21) is the use of θ_i vs θ_t. The only way to compute μ_t^* exactly is to recompute the entire sum after every parameter update. Alternatively, we could somehow approximate this sum. Below we will define $g_i^t = \nabla_{\theta_t} J_i(\theta_t)$, which we will then approximate with $\hat{g}_i^t$. We will then show that this approximation has a recursive form which leads to an algorithm.

We want to correct accumulated past gradients g_i to their "ideal" form g_i^t, as above. We do so with their Taylor expansion around θ, and we write the correction of the gradient g_i computed at learning time i corrected at time t as:

$$g_i^t = g_i(\theta_i + \Delta\theta(t-1;i)) = g_i + \nabla_\theta g_i^T \Delta\theta(t-1;i) + o(\|\Delta\theta\|_2^2) \tag{4.22}$$

$$\approx \hat{g}_i^t = g_i + Z_i^T \Delta\theta \tag{4.23}$$

where $\Delta\theta(t;i) = \theta_t - \theta_i$, $g_i = \nabla_{\theta_i} J_i$, $Z_i = \nabla_{\theta_i} g_i$. Note that we allow ourselves this simplification knowing that the parameter space of a deep network is filled with locally consistent regions in which Taylor expansions are accurate (Balduzzi et al., 2017).

Here we are agnostic of the particular form of Z, which will depend on the loss and learning algorithm, and is *not necessarily* the so-called Hessian. To see why this is the case, and for the derivation of Z for the squared loss, cross-entropy and TD(0), see §4.2.3.1.

Let's now express $\hat{g}_i^t$ in terms of g_i and updates μ_t. At each learning step, the current momentum μ_t is multiplied with the learning rate α to update the parameters, which allows us to more precisely write $\hat{g}_i^t$:

$$\theta_t = \theta_{t-1} - \alpha\mu_t = \theta_0 - \alpha \sum_{i=1}^{t} \mu_i \tag{4.24}$$

$$\Delta\theta(t;i) = \theta_t - \theta_i = \theta_0 - \alpha \sum_{k=1}^{t} \mu_k - \theta_0 + \alpha \sum_{k=1}^{i} \mu_k = -\alpha \sum_{k=i}^{t} \mu_k \tag{4.25}$$

$$\hat{g}_i^t = g_i + Z_i^T \Delta\theta(t-1;i) = g_i - \alpha Z_i^T \sum_{k=i}^{t-1} \mu_k \tag{4.26}$$

We can now write $\hat{\mu}_t$, the approximated μ_t^* using $\hat{g}_i^t$:

$$\begin{aligned}
\hat{\mu}_t &= (1-\beta)g_t + (1-\beta)\sum_{i=1}^{t-1} \beta^{t-i} \hat{g}_i^t && (4.27)\\
&= (1-\beta)g_t + (1-\beta)\sum_{i=1}^{t-1} \beta^{t-i} g_i - (1-\beta)\sum_{i=1}^{t-1} \beta^{t-i} \alpha Z_i^T \sum_{k=i}^{t-1} \hat{\mu}_k && (4.28)\\
&= \mu_t - \alpha(1-\beta)\sum_{i=1}^{t-1}\sum_{k=i}^{t-1} \beta^{t-i} Z_i^T \hat{\mu}_k \quad \text{extract } \mu && (4.29)\\
&= \mu_t - \alpha(1-\beta)\sum_{k=1}^{t-1}\sum_{i=1}^{k} \beta^{t-i} Z_i^T \hat{\mu}_k \quad \text{change the sum indices for convenience} && (4.30)\\
&= \mu_t - \eta_t \;\; \text{extract } \eta_t && (4.31)
\end{aligned}$$

Note that here we plugged in $\hat{\mu}_k$ rather than μ_k, this is because we defined $\hat{g}_i^t$ in terms of *some momentum updates* μ_k, without specifying what those were. Since we're now using $\hat{\mu}_k$ momentum updates, we replace $\hat{g}_i^t$ above as a function of $\hat{\mu}_k$.

Let's try to find a recursive form for η_t:

$$\begin{aligned}
\eta_t - \eta_{t-1} &= \alpha(1-\beta)\sum_{k=1}^{t-1}\sum_{i=1}^{k} \beta^{t-i} Z_i^T \hat{\mu}_k - \alpha(1-\beta)\sum_{k=1}^{t-2}\sum_{i=1}^{k} \beta^{t-1-i} Z_i^T \hat{\mu}_k\\
&= \alpha(1-\beta)\left(\sum_{k=1}^{t-2}\sum_{i=1}^{k}(\beta^{t-i} Z_i^T \hat{\mu}_k - \beta^{t-1-i} Z_i^T \hat{\mu}_k) + \sum_{i=1}^{t-1} \beta^{t-i} Z_i^T \hat{\mu}_{t-1}\right)\\
&= \alpha(1-\beta)\left(\sum_{k=1}^{t-2}\sum_{i=1}^{k}(\beta-1)\beta^{t-1-i} Z_i^T \hat{\mu}_k + \sum_{i=1}^{t-1} \beta^{t-i} Z_i^T \hat{\mu}_{t-1}\right)\\
&= (\beta-1)\eta_{t-1} + \alpha\beta(1-\beta)\sum_{i=1}^{t-1} \beta^{t-1-i} Z_i^T \hat{\mu}_{t-1}\\
\text{Let} \qquad & \zeta_t = (1-\beta)\sum_{i=1}^{t} \beta^{t-i} Z_i\\
\eta_t - \eta_{t-1} &= (\beta-1)\eta_{t-1} + \alpha\beta\zeta_{t-1}\hat{\mu}_{t-1}\\
\eta_t - \eta_{t-1} &= -(1-\beta)\eta_{t-1} + \alpha\beta\zeta_{t-1}^T\hat{\mu}_{t-1}\\
\eta_t &= \beta\eta_{t-1} + \alpha\beta\zeta_{t-1}^T\hat{\mu}_{t-1}
\end{aligned}$$

We can now write the full update as:

$$\begin{aligned}
\hat{\mu}_t &= \mu_t - \eta_t \\
\eta_t &= \beta\eta_{t-1} + \alpha\beta\zeta_{t-1}^T\hat{\mu}_{t-1} \\
\mu_t &= (1-\beta)\sum_{i=1}^{t}\beta^{t-i}g_i = (1-\beta)g_t + \beta\mu_{t-1} \\
\zeta_t &= (1-\beta)\sum_{i=1}^{t}\beta^{t-i}Z_i = (1-\beta)Z_t + \beta\zeta_{t-1}
\end{aligned}$$

with $\eta_0 = \mathbf{0}$.

This corresponds to an algorithm where one maintains η, μ and ζ.

4.2.3.1 Derivation of Taylor expansions and Z

For least squares regression, the expansion around θ of $g(\theta)$ is simply the Hessian of the loss δ^2:

$$\delta^2 = \frac{1}{2}(y - f_\theta(x))^2 \tag{4.32}$$

$$g(\theta) = \nabla_\theta \delta^2 \tag{4.33}$$

$$g(\theta + \Delta\theta) = g(\theta) + H_J^T \Delta\theta + o(\|\Delta\theta\|_2^2) \tag{4.34}$$

This Hessian has the form

$$H_J = \nabla_\theta f \otimes \nabla_\theta f + \delta^2 \nabla_\theta^2 f \tag{4.35}$$

δ^2 being small when a neural network is trained and ∇^2 being expensive to compute, a common approximation to H_j is to only rely on the outer product. Thus we can write:

$$Z_{reg} = \nabla_\theta f \otimes \nabla_\theta f \tag{4.36}$$

For classification, or categorical crossentropy, Z has exactly the same form, but where f is the log-likelihood (i.e. output of a log-softmax) of the correct class.

For TD(0), the expansion is more subtle. Since TD is a semi-gradient method, when computing $g(\theta)$, gradient of the TD loss δ^2, we ignore the bootstrap target's derivative, i.e. we hold $V_\theta(s')$ constant (unlike in GTD). On the other hand, when computing the Taylor expansion around θ, we do care about how $V_\theta(s')$ changes, and so its gradient comes into play:

$$
\begin{aligned}
g(\theta) &= (V_\theta(x) - \gamma V_\theta(x') - r)\nabla_\theta V_\theta(x) \\
g(\theta + \Delta\theta)_i &= g_i(\theta) + (\nabla_\theta V_\theta(x) - \gamma \nabla_\theta V_\theta(x'))\nabla_{\theta_i} V_\theta(x) \cdot \Delta\theta + \delta \nabla_\theta \nabla_{\theta_i} V_\theta(x) \cdot \Delta\theta \\
g(\theta + \Delta\theta) &= g(\theta) + ((\nabla_\theta V_\theta(x) - \gamma \nabla_\theta V_\theta(x')) \otimes \nabla_\theta V_\theta(x))^T \Delta\theta + \delta \nabla^2_\theta V_\theta(x)^T \Delta\theta
\end{aligned}
$$

Similarly for TD we ignore the second order derivatives and write:

$$
Z_{TD} = (\nabla_\theta V_\theta(x) - \gamma \nabla_\theta V_\theta(x')) \otimes \nabla_\theta V_\theta(x)
$$

For an n-step TD objective, the correction is very similar:

$$
Z_{TD(n)} = (\nabla_\theta V_\theta(x_t) - \gamma^n \nabla_\theta V_\theta(x_{t+n})) \otimes \nabla_\theta V_\theta(x_t)
$$

For a forward-view TD(λ) objective this is also similar, but more expensive:

$$
Z_{TD(\lambda)} = (\nabla_\theta V_\theta(x_t) - (1-\lambda)\nabla_\theta(\gamma\lambda V_\theta(x_{t+1}) + \gamma^2\lambda^2 V_\theta(x_{t+2}) + ...)) \otimes \nabla_\theta V_\theta(x_t)
$$

Finally, to avoid maintaining $n \times n$ values for Z, it is possible to only maintain the diagonal of Z or some block-diagonal approximation, at some performance cost.

4.2.3.2 Additional Remarks

Much of RL analysis is done in the linear case, as it is easier to make precise statements about convergence there. We write down such a case below, but we were unfortunately unable to derive any interesting analyses from it.

Recall that the proposed method attempts to approximate μ^*:

$$
\mu^*_t = (1-\beta)\sum_{i=1}^{t} \beta^{t-i} \nabla_{\theta_t} J_i(\theta_t)
$$

which in the linear case $V_\theta(x) = \theta^\top \phi(x)$ is simply:

$$
= (1-\beta)\sum_{i=1}^{t} \beta^{t-i} \delta_i \phi_i
$$

which depends on θ_i through δ_i the TD error. As such we can write Z as:

$$
Z_{TD} = (\nabla_\theta V_\theta(x) - \gamma \nabla_\theta V_\theta(x')) \otimes \nabla_\theta V_\theta(x) = (\phi - \gamma\phi')\phi^\top
$$

which interestingly does not depend on θ. To the best of our knowledge, and linear algebra skills, this lack of dependence on θ does *not* allow for a simplification of the proposed correction mechanism (in contrast with linear eligibility traces) due to the pairwise multiplicative t, i dependencies that emerge.

As for similar adaptive gradient methods (Romoff et al., 2021), we require per-input gradients. Fortunately, these can be computed efficiently for a minibatch of examples with methods such as BackPack (Dangel et al., 2020).

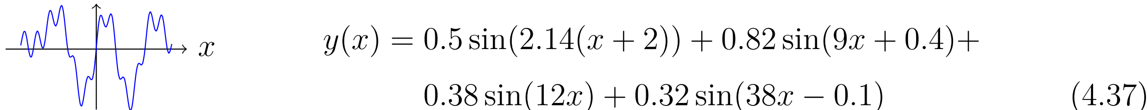
4.3 Empirical Results

We evaluate our proposed correction as well as the oracle (the best our method could perform) on common RL benchmarks and supervised problems. We evaluate each method with a variety of hyperparameters and multiple seeds for each hyperparameter setting. The full range of hyperparameters used as well as architectural details can be found in §4.3.3.

We will use the following notation for the optimizer used to train models: μ is the usual momentum, i.e. (4.2)&(4.3), and serves as a baseline. μ^* is our oracle, defined in (4.6). $\hat{\mu}$ is the proposed correction, with updates as in (4.16)-(4.19), using the outer product approximation of Z. $\hat{\mu}_{\text{diag}}$ is the proposed correction, but with a diagonal approximation of Z. Throughout our figures, shaded areas are bootstrapped 95% confidence intervals over hyperparameter settings (if applicable) and runs.

4.3.1 Supervised Learning

We first test our hypothesis, that there is bias in momentum, in a simple regression task and SVHN (Netzer et al., 2011). For regression, we task a 4-layer MLP to regress to a 1-d function from a 1-d input. For illustration we use a smooth but non-trivial mixture of sines of increasing frequency in the domain $x \in [-1, 1]$:



$$y(x) = 0.5\sin(2.14(x+2)) + 0.82\sin(9x+0.4)+ \\ 0.38\sin(12x) + 0.32\sin(38x-0.1) \tag{4.37}$$

Note that this choice is purely illustrative and that our findings extend to similar simple functions. We train the model on a fixed sample of 10k uniformly sampled points. We measure the mean squared error (MSE) when training a small MLP with momentum SGD versus our oracle momentum μ^* and the outer product correction $\hat{\mu}$. As shown in Figure 4.1, we find that while there is a significant difference between the oracle and the baseline ($p \approx 0.001$ from Welsh's t-test), the difference is fairly small, and is no longer significant when using the corrected $\hat{\mu}$ ($p \approx 0.1$).

We compare training a small convolutional neural network on SVHN (Netzer et al., 2011) with momentum SGD versus our oracle momentum μ^* and the diagonalized correction momentum $\hat{\mu}_{\text{diag}}$. The results are shown in Figure 4.2. We do not find significant

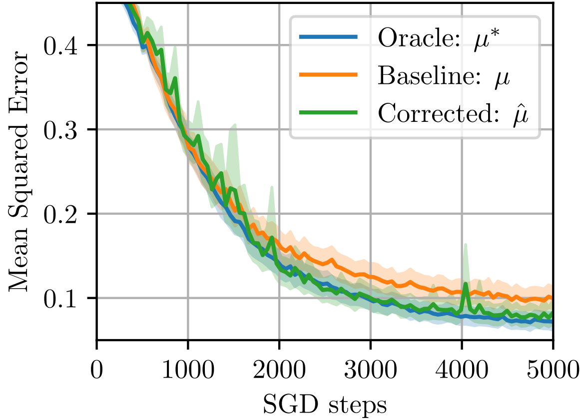


Figure 4.1: Regression to (4.37) with varying momentums (10 seeds per setting). Note that the difference between μ and $\hat{\mu}$ is not significant, but μ and μ^* is.

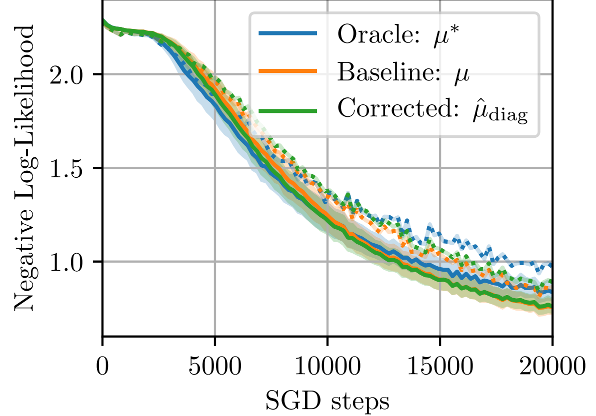


Figure 4.2: Classification on SVHN with varying momentums (5 seeds per setting). Dotted lines are test losses. The only significant difference is between the training loss of μ^* and μ.

differences except between the training loss of μ^* and μ, and find that the oracle performs *worse* than the normal and corrected momentum.

From these experiments we conclude that, in supervised learning, there exists a quantifiable optimization bias to momentum, but that correcting it does not appear to offer any benefit. It improves performance only marginally at best, and degrades it at worst. This is consistent with the insight that μ^* approximates the large batch gradient, and that large batches are often associated with overfitting or poor convergence in supervised learning (Wilson and Martinez, 2003; Keskar et al., 2017b).

4.3.2 Temporal Difference Learning

We now test our hypotheses, that there is optimisation bias and that we can correct it, on RL problems. First, we test policy evaluation of the optimal policy on the Mountain Car problem (Singh and Sutton, 1996) with a small MLP. We also test the standard Acrobot and Cartpole environments (Sutton and Barto, 2018) and find very similar results. We then test our method on Atari (Bellemare et al., 2013) with convolutional networks.

4.3.2.1 Testing the method on standard problems

Figure 4.3 shows policy evaluation on Mountain Car using a replay buffer (on-policy state transitions are sampled i.i.d. in minibatches). We compare the loss distributions (across hyperparameters and seeds) at step 5k, and find that all methods are significantly different ($p < 0.001$) from one another. Figure 4.4 shows online policy evaluation, i.e. the transitions

are generated and learned from once, in-order, and one at a time (minibatch size of 1). There we see that the oracle μ^* and full corrected version $\hat{\mu}$ are significantly different from the baseline μ ($p < 0.001$) and diagonalized correction $\hat{\mu}_{\text{diag}}$, as well as μ from $\hat{\mu}_{\text{diag}}$, while μ^* and $\hat{\mu}$ are not significantly different ($p > 0.1$).

This suggests that the ζ matrix carries useful off-diagonal temporal information about parameters which co-vary, especially when the data is not used uniformly during learning. We test another possible explanation, which is that performance is degraded in online learning because the batch size is 1 (rather than 16 or 32 as in Figure 4.3). We find that a batch size of 1 does degrade $\hat{\mu}_{\text{diag}}$'s performance significantly, as shown in Figure 4.6, but does not fully explain its poor online performance.

In Figure 4.7, we evaluate the effect of the momentum parameter β, and in Figure 4.8 we evaluate the effect of the learning rate. Both show the expected U-shaped curves, where there appears to be fairly clear optimal hyperparameter regions. We additionally test our method on policy evaluation in Acrobot and Cartpole, which are two standard RL problems. Results are shown in Figures 4.9 and 4.10.

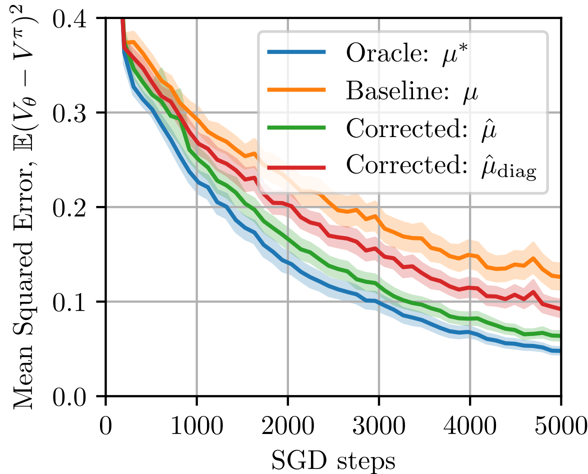


Figure 4.3: TD(0) policy evaluation on Mountain Car with varying momentums on a **replay buffer**. The MSE is measured against a pretrained V^π (10 seeds per setting). At step 5k, all methods are significantly different.

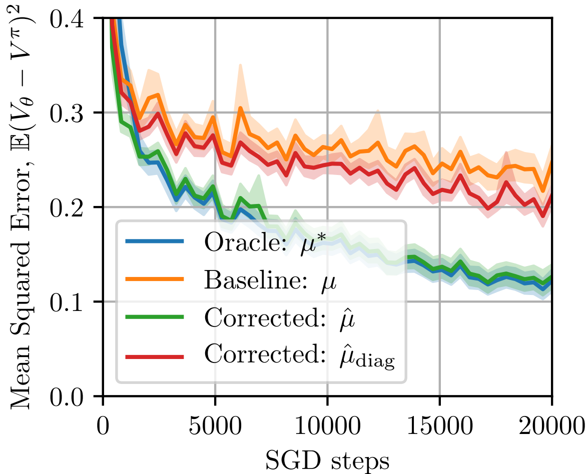


Figure 4.4: TD(0) **online** policy evaluation on Mountain Car, transitions are seen in-order. The MSE is measured against a pre-trained V^π (50 seeds per setting). At step 20k, μ^* and $\hat{\mu}$ are not significantly different.

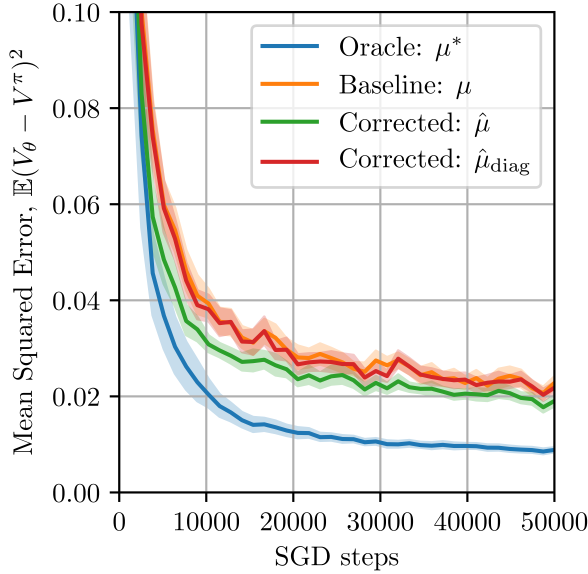


Figure 4.5: Replication of Figure 4.3 with 10× more training steps. Methods gradually converge to the value function.

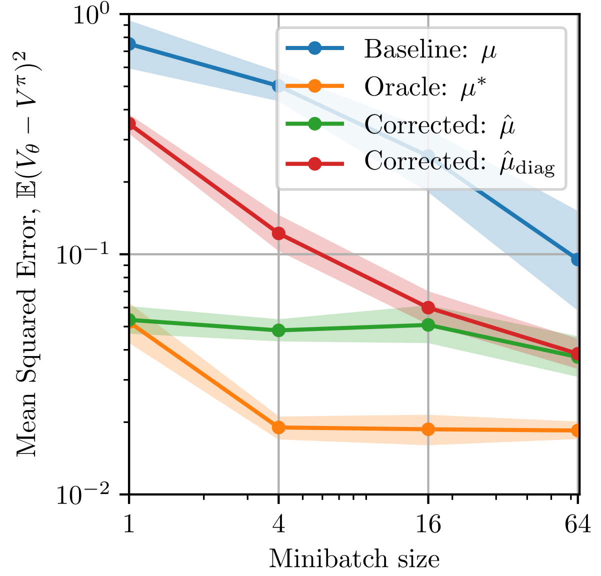


Figure 4.6: TD(0) policy evaluation on Mountain Car with varying minibatch size on a **replay buffer**. The MSE is measured after 5k SGD steps against a pretrained V^π. Shaded areas are bootstrapped 95% confidence runs (20 seeds per setting).

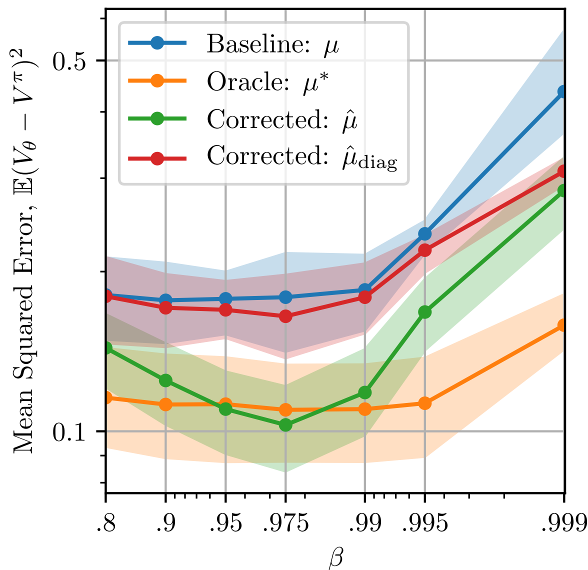


Figure 4.7: TD(0) policy evaluation on Mountain Car with varying β on a **replay buffer**. The MSE is measured after 5k SGD steps against a pretrained V^π. Shaded areas are bootstrapped 95% confidence runs (10 seeds per setting). We use a minibatch size of 4 to reveal interesting trends.

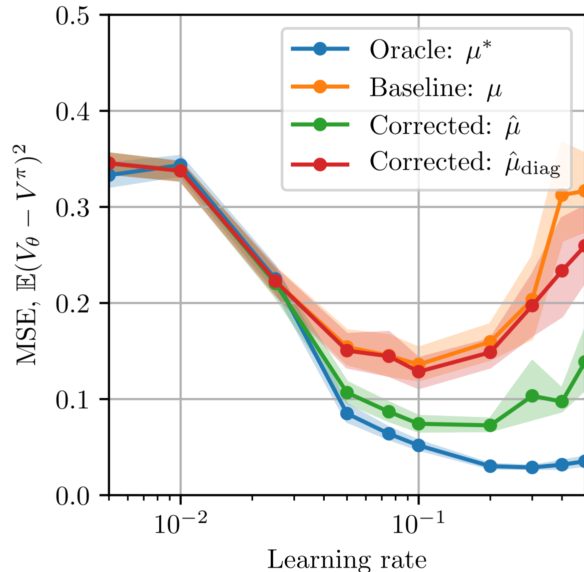


Figure 4.8: Effect of the learning rate on Mountain Car, replay buffer policy evaluation, MSE after 5k training steps. The MSE is measured after 5k SGD steps against a pretrained V^π. Shaded areas are bootstrapped 95% confidence runs (20 seeds per setting).

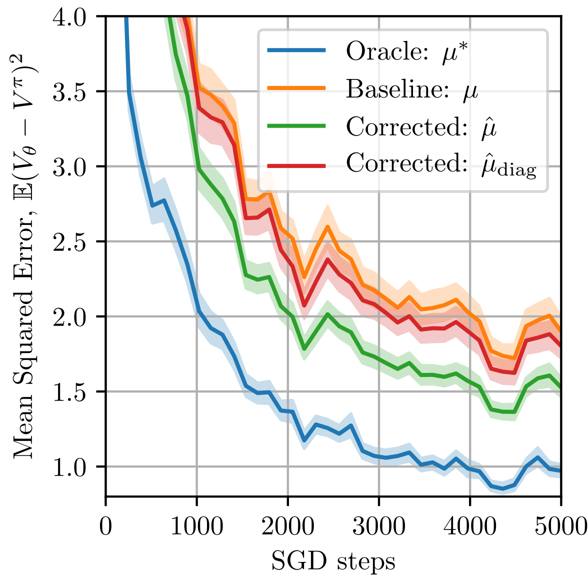


Figure 4.9: TD(0) policy evaluation on Acrobot with varying hyperparameters on a **replay buffer**. The MSE is measured after 5k SGD steps against a pretrained V^π. Shaded areas are bootstrapped 95% confidence runs (10 seeds per setting).

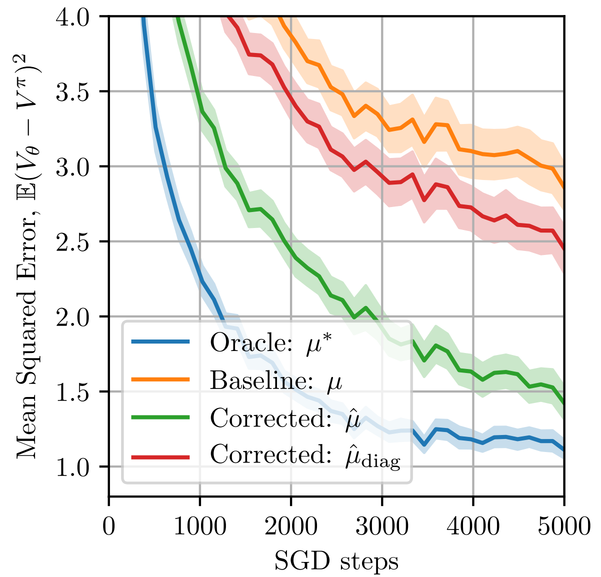


Figure 4.10: TD(0) policy evaluation on Cartpole with varying hyperparameters **replay buffer**. The MSE is measured after 5k SGD steps against a pretrained V^π. Shaded areas are bootstrapped 95% confidence runs (10 seeds per setting).

4.3.2.2 Understanding the momentum correction

A central motivation of this work is that staleness in momentum arises from the change in gradients and targets. In theory, this is especially problematic for function approximators which tend to have interference, such as DNNs (Fort et al., 2019; Bengio et al., 2020a), i.e. where taking a gradient step using a single input affects the output for virtually every other input. More interference means that as we change θ, the accumulated gradients computed from past θs become stale faster. We test this hypothesis by (1) measuring the *drift* of the value function (how much $V(s)$ changes over learning steps for some fixed s) in different scenarios, and (2) computing the cosine similarity between the corrected gradients of (4.13), $\hat{g}_i^t$, and their true value $g_i^t = \nabla J_i(\theta_t)$.

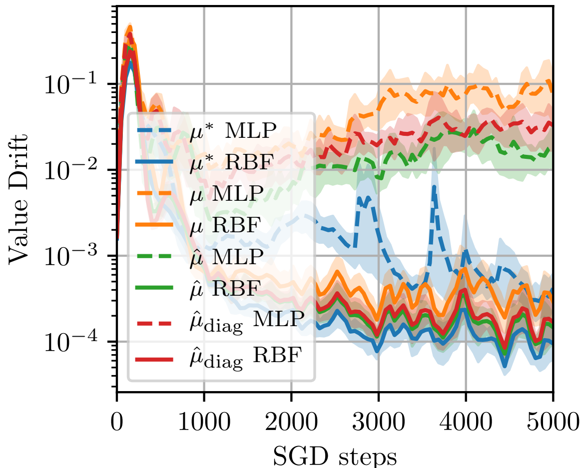


Figure 4.11: **Value drift** of $V(s')$ when training with TD(0) on a replay buffer. We see that RBFs being a sparse feature representation, the value functions of recently seen data tend not to drift (10 seeds per setting). Here $\sigma^2 = 1$.

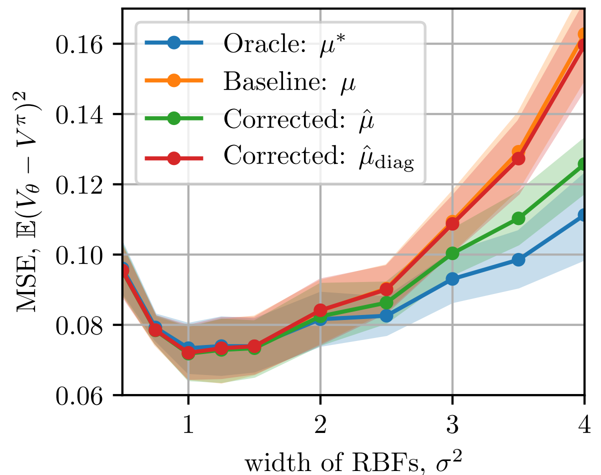


Figure 4.12: MSE as a function of the **width**, σ^2, of RBF kernels. The larger the kernel, the more value drift our method, $\hat{\mu}$, is able to correct (10 seeds per setting).

In Figure 4.11 we compare the value drift of MLPs with that of linear models with Radial Basis Function (RBF) features. We compute the value drift of the target on recently seen examples, i.e. we compute the average $(V_{\theta_i}(s_i') - V_{\theta_t}(s_i'))^2$ for the last $h = 2n_{mb}/(1-\beta)$ examples, where n_{mb} is the minibatch size. We compute the RBF features for $s \in \mathbb{R}^2$ as $\exp(-\|s - u_{ij}\|^2/\sigma^2)$ for a regular grid of $u_{ij} \in \mathbb{R}^2$ in the input domain. We find that the methods we try (even the oracle) are all virtually identical when using RBFs. We also find, as shown in Figure 4.11, that RBFs have very little value drift (due to their sparsity) compared to MLPs. This is consistent with our hypothesis that the method we propose is only useful if there is value drift–otherwise, there is no **optimization bias** incurred by

using momentum. We can test this hypothesis further by artificially increasing the width of the RBFs, σ^2, such that they overlap. As predicted, we find that reducing sparsity (increasing interference) increases value drift and increases the gap between our method and the baseline (Figure 4.12). This drift correlates with performance when changing σ^2 (Figure 4.13).

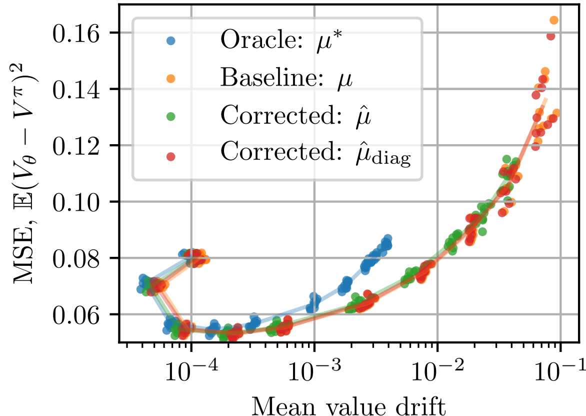


Figure 4.13: MSE as a function of mean **Value drift** of $V(s')$ for RBFs of varying kernel size. The lines match the σ^2 lines of Figure 4.12, and show the relation between σ^2, drift, and error.

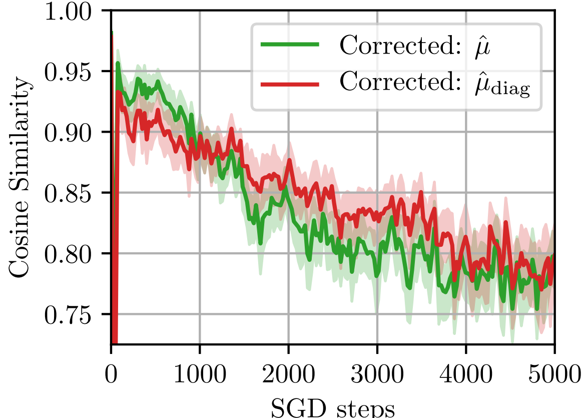


Figure 4.14: Average **cosine similarity** of the Taylor approximations $\hat{g}_i^t$ with their true value g_i^t for recently seen data; Mountain Car, replay buffer policy evaluation (40 seeds per setting).

In Figure 4.14 we measure the cosine similarity of the Taylor approximations $\hat{g}_i^t = g_i + Z_i^\top(\theta_t - \theta_i)$ with the true gradients $g_i^t = \nabla J_i(\theta_t)$ for the last $h = 2n_{mb}/(1-\beta)$ examples. We find that the similarity is relatively high (close to 1) throughout training but that it gets lower as the model converges to the true value V^π. This is also consistent with our hypothesis that there is change (staleness) in gradients, while also validating the approach of using Taylor approximations to achieve this correction mechanism.

It is also possible to correct momentum for a subset of parameters; we try this on a per-layer basis and find that, perhaps counter-intuitively, it is *much* better to correct the bottom layers (close to x) than the top layers (close to V), and correcting all layers is the best (see Figure 4.15). Although one may expect that changes close to V should produce more drift (it is commonly thought that bootstrapping happens in the last layer on top of relatively stable representations), the opposite is consistent with $\hat{\mu}$ interacting with *interference* in the input space, which the bottom layers have to learn to disentangle.

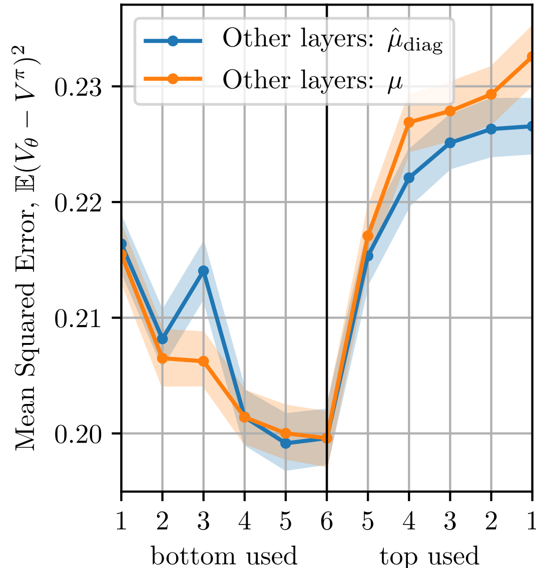


Figure 4.15: TD(0) policy evaluation on Mountain Car with an MLP. We vary the number of layers whose parameters are used for full $\hat{\mu}$ correction (n^2 params); e.g. when "bottom used" is 3, the first 3 layers, those closest to the input, are used; when "top used" is 1, only the last layer, that predicts V from embeddings, is used. The parameters of other layers are either corrected with the diagonal correction or use normal momentum. Correcting "both ends" is not better than just the bottom (not shown here).

4.3.2.3 Correcting momentum in Atari

We now apply our method on the Atari MsPacman game. We first do **policy evaluation** on an expert agent (we use a pretrained Rainbow agent (Hessel et al., 2018)). Since the architecture required to train an Atari agent is too large ($n \approx 4.8$M) to maintain n^2 values, we only use the diagonal version of our method. We also experiment with smaller ($n \approx 12.8$k) models and the full correction, with similar results (see §4.3.3.8). To (considerably) speed up learning, since the model is large, we additionally use per-parameter learning rates as in Adam (Kingma and Ba, 2015), where an estimate of the second moment is used as denominator in the parameter update; we denote this combination $\hat{\mu}_{\rm diag}/\sqrt{\nabla^2 + \epsilon}$. We see in Figure 4.16 that our method provides a significant ($p < 0.01$) advantage.

Note that our method does not use frozen targets (as is usually necessary for this environment). A simple way to avoid momentum staleness and/or target drift in TD(0) is the use of **frozen targets**, i.e. to keep a separate $\bar{\theta}$ to compute $V_{\bar{\theta}}(s')$, updated ($\bar{\theta} \leftarrow \theta$) at large intervals. Such a method is central to DQN (Mnih et al., 2013), but its downside is that it requires more updates to bootstrap. We find that for *policy evaluation*, frozen targets are much slower (both in Atari and simple environments) than our baseline (see Figures 4.17 and 4.18).

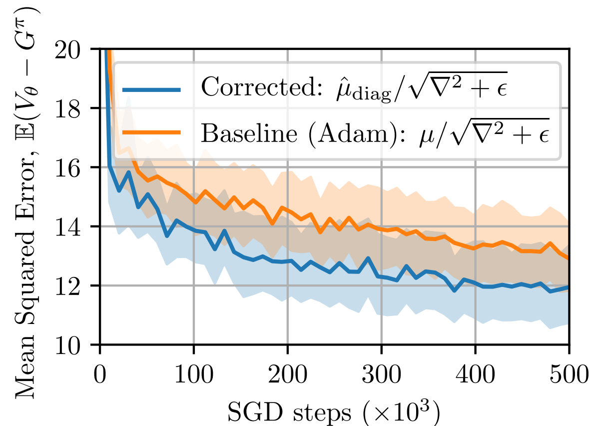


Figure 4.16: TD(0) policy evaluation on Atari (MsPacman) with varying momentums (20 seeds) on a **replay buffer**. The MSE is measured against sampled returns G^π.

We finally apply our method to **control**, first in combination with a **policy gradient** method, PPO (Schulman et al., 2017), in its policy evaluation part, and second with a 5-step **Sarsa** method, but find no improvement (or marginal at best) in either setting. As in simpler environments we measure cosine similarity and value drift. We find low similarity (≈ 0.1) but a $2\times$ to $3\times$ decrease in drift using our method, suggesting that while our method corrects drift, its effect on policy improvement is minor. We suspect that in control, even with a better value function, other factors such as exploration or overestimation come into play which are not addressed by our method.

4.3.3 Hyperparameters and Architecture Choices

All experiments are implemented using PyTorch (Paszke et al., 2019b). We use Leaky ReLUs throughout. All experimental code is available at `https://github.com/bengioe/staleness-corrected-momentum`.

On Leaky ReLUs: we did experiment with ReLU, tanh, ELU, and SELU activation units. The latter 3 units have more stable Taylor expansions for randomly initialized neural networks, but in terms of experimental results, Leaky ReLUs were always significantly better.

4.3.3.1 Regression

We use an MLP with 4 layers of width n_h.

We use the cross-product of $n_h \in \{8, 16, 32\}$, $\beta \in \{0.9, 0.99\}$, $\alpha \in \{0.005, 0.01\}$, $n_{mb} \in \{4, 16, 64\}$.

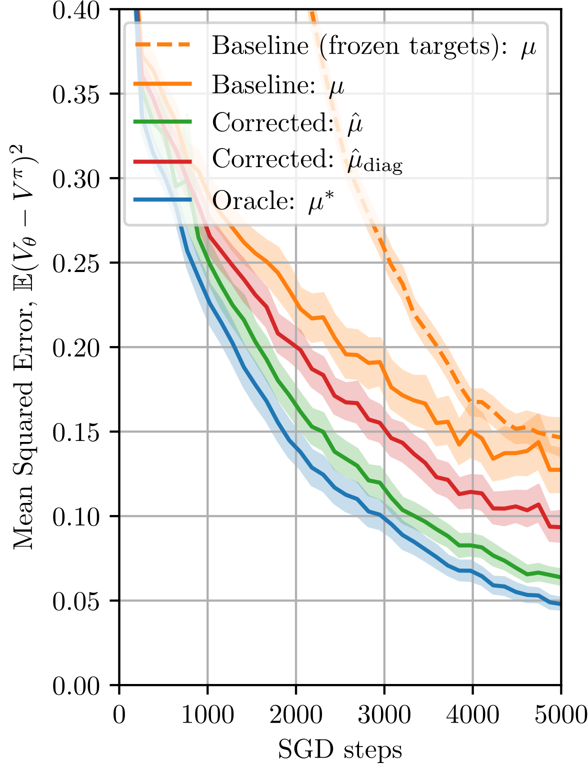


Figure 4.17: Replication of Figure 4.3 including the frozen targets baseline.

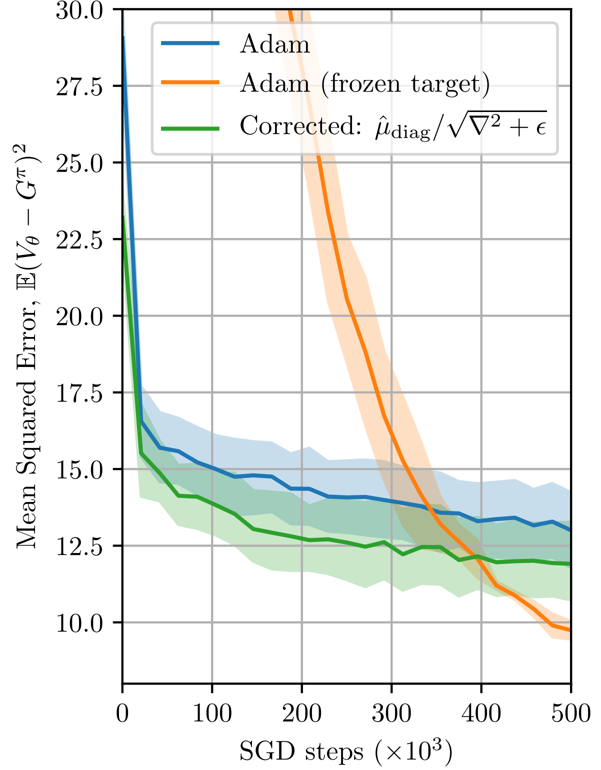


Figure 4.18: Replication of Figure 4.16 including the frozen targets baseline. Interestingly the models trained with frozen targets eventually become more precise than those without, but this only happens after a very long time. This is explained by the stability required for bootstrapping when TD errors become increasingly small, which is easily addressed by keeping the target network fixed.

4.3.3.2 SVHN

We use a convolutional model with the following sequence of layers, following PyTorch convention: Conv2d(3, n_h, 3, 2, 1), Conv2d(n_h, $2n_h$, 3, 2, 1), Conv2d($2n_h$, $2n_h$, 3, 2, 1), Conv2d($2n_h$, n_h, 3, 1, 1), Flatten(), Linear($16n_h$, $4n_h$), Linear($4n_h$, $4n_h$), Linear($4n_h$, 10), with LeakyReLUs between each layer.

We use the cross-product of $n_h \in \{8, 16\}$, $\beta \in \{0.9, 0.99\}$, $\alpha \in \{0.005, 0.01\}$, $n_{mb} \in \{4, 16, 64\}$.

4.3.3.3 Mountain Car

We use an MLP with 4 layers of width n_h.

We use the cross-product of $n_h \in \{8, 16, 32\}$, $\beta \in \{0.9, 0.99\}$, $\alpha \in \{0.5, 0.1, 0.05\}$, $n_{mb} \in \{4, 16, 64\}$.

4.3.3.4 Mountain Car online

We use an MLP with 4 layers of width n_h.

We use the cross-product of $n_h \in \{16\}$, $\beta \in \{0.9, 0.99\}$, $\alpha \in \{0.005, 0.001, 0.0005\}$, $n_{mb} \in \{1\}$.

4.3.3.5 Value Drift

We use a linear layer on top of a grid RBF representation with each gaussian having a variance of σ^2/n_{grid}.

For the RBF we use $n_{grid} = 20$, $\alpha = 0.1$, $n_{mb} = 16$, $\beta = 0.99$. For the MLP we use $n_h = 16$, $\alpha = 0.1$, $n_{mb} = 16$, $\beta = 0.99$.

4.3.3.6 RBF value drift ranges

We use RBFs with $\sigma^2 \in \{4, 3.5, 3, 2.5, 2, 1.5, 1.25, 1, 0.75, 0.5\}$, $n_{grid} \in \{10, 20\}$, $\alpha \in \{0.1, 0.01\}$.

4.3.3.7 Cosine Similarity

We use an MLP with 4 layers of width $n_h = 16$, $\alpha = 0.1$, $n_{mb} = 16$, $\beta = 0.95$.

4.3.3.8 Atari

We use the convolutional model of Mnih et al. (2013) with the same default hyperparameters, following PyTorch convention: Conv2d(4, n_h, 8, stride=4, padding=4), Conv2d(n_h, $2n_h$, 4, stride=2, padding=2), Conv2d($2n_h$, $2n_h$, 3, padding=1), Flatten(), Linear($2n_h \times 12 \times 12$, $16n_h$), Linear($16n_h$, n_{acts}).

We use $n_h = 64$, $n_{mb} = 32$, for Adam we use $\alpha = 5 \times 10^{-5}$ and $\beta = 0.99$, for our method we use $\alpha = 10^{-4}$ and $\beta = 0.9$ (these choices are the result of a minor hyperparameter search of which the best values were picked, equal amounts of compute went towards our method and the baseline so as to avoid "poor baseline cherry picking"). For the frozen target baseline we update the target every 2.5k steps.

We were able to find similar differences between Adam and our method with a much smaller model, but using the full correction instead of the diagonal one. Although the full correction outperforms Adam when both use this small model, using so few parameters is not as accurate as the original model described above, and we omit these results.

This small model is: Conv2d(4, n_h, 3, 2, 1), Conv2d(n_h, n_h, 3, 2, 1), Conv2d(n_h, n_h, 3, 2, 1), Conv2d(n_h, n_h, 3, 2, 1), Conv2d(n_h, n_{acts}, 6). For $n_h = 16$ (which we used for experiments), this model has a little less than 16k parameters, making Z about 250M scalars. While this is large, this easily fits on a modern GPU, and the extra computation time required for the full correction mostly still comes from computing two extra backward passes, rather than from computing Z and the correction.

This is beyond the scope of this thesis, but seems worth of note: Interestingly this small model still works quite well for control (both with Adam and our method). We have not tested this extensively, but, perhaps contrary to popular RL wisdom surrounding Deep Q-Learning, we were able to train decent MsPacman agents with (1) no replay buffer, but rather 4 or more parallel environments (the more the better) as in A2C (Mnih et al., 2016) (2) no frozen target network (3) a Q network with only 16k parameters rather than the commonplace 4.8M-parameter Atari DQN model. The only "trick" required is to use 5-step Sarsa instead of 1-step TD (as suggested by the results of Fedus et al. (2020), although in their case a replay buffer is used).

4.3.3.9 Mountain Car minibatch size

We use the same configuration than previously with $n_{mb} \in \{1, 4, 16, 64\}$.

4.3.3.10 Mountain Car momentum parameter

We use the same configuration than previously with $\beta \in \{.8, .9, .95, .975, .99, .995, .999\}$.

4.3.3.11 Mountain Car MLP depth

We use an MLP with 6 layers of width $n_h = 16$, $\alpha = 0.1$, $n_{mb} = 16$, $\beta = 0.9$. We vary which layers get used in the corrected momentum; see caption.

4.3.3.12 Acrobot and Cartpole

We use the same settings as for Mountain Car, with the exception that we do 5-step TD for Acrobot, and 3-step TD for Cartpole. Using $n > 1$ appears necessary for convergence for both our method and the baseline.

4.4 Discussion

So far in this thesis we've discussed generalization and memorization showing that good models arise from a balance of the two, and linked generalization and interference by show-

ing that Temporal Difference learning pushes DNNs to memorize too much as a result of poor optimization dynamics. We then proposed a novel method to change those dynamics by taking into account the non-stationary nature of the TD objective.

We found that this method improves momentum, when applied to DNNs doing TD learning, by correcting gradients for their staleness via an approximate Taylor expansion. We showed that correcting this staleness is particularly useful when learning online using a TD objective, but less so for supervised learning tasks. We showed that the proposed method corrects for value drift in the bootstrapping target of TD, and that the proposed approximate Taylor expansion is a useful tool that aligns well enough with the true gradients.

4.4.1 Limitations

In its most principled form, the proposed method requires computing a second-order derivative, which is impractical in most deep learning settings. While we do find experimentally that ignoring its contribution has a negligible effect in toy settings, we are unable to verify this for larger neural networks. Compared to the usual momentum update, the proposed method also requires performing two additional backward passes, as well as storing $2n+n^2$ additional scalars. This can be improved with a diagonal approximation requiring only $3n$ scalars, but this approximation is not as precise, which is currently an obstacle for large architectures.

While these extra computations improve convergence speed, since we get more out of each sample, they do not necessarily improve convergence speed in *computation time* because of the n^2 scaling. Nonetheless, since this method is inherently parallel, it does scale relatively well with parallel computing. For example, our method is suited to GPU parallelism and has reasonable speed even for large n^2s, even in Atari settings.

One meta-result stands out from this work: something is lost when naively applying supervised learning optimization tools to RL algorithms. In particular here, by simply taking into account the non-stationarity of the TD objective, we successfully improve the per-update learning speed of a standard tool (momentum), and demonstrate the potential value of incorporating elements of RL into supervised learning tools. The difference explored here with momentum is only one of the many differences between RL and supervised learning, and there remain plenty of opportunities to improve deep learning methods by understanding their interaction with the peculiarities of RL.

4.4.2 The Supremacy of Accelerated Adaptive Methods

So far we've mostly discussed so-called accelerated methods, of which momentum is the canonical form. These methods maintain some form of velocity, pushing parameters at a certain speed which only varies a little at each learning iteration.

Adaptive methods are another important class of methods, instead of maintaining a velocity, they maintain a per-parameter learning rate. The canonical adaptive method in deep learning is RMSprop (Tieleman and Hinton, 2012), which uses the inverse approximate standard deviation of each parameter's derivative as an adaptive learning rate.

Adam (Kingma and Ba, 2015) combines these two ideas into one, and is important to single out as it currently sits on top of optimization methods used in deep RL (Henderson et al., 2018; Ceron and Castro, 2021). There exists a large variety of optimizers for supervised learning, yet none seem to be able to significantly improve performance in deep RL beyond that of Adam. The accelerated method proposed in this chapter does not reliably scale to complex high-dimensional problems. Another recent attempt (to which I made a minor code contribution) by Romoff et al. (2021), which proposes an adaptive optimizer for TD, also struggles to reliably improve performance.

All these recent results seem to suggest that not much can be done in terms of naive optimization for 1-step bootstrapping methods like TD(0). Indeed, our own results show how TD(λ) and n-step methods provide more stable targets from which it is faster to learn (see also Fedus et al., 2020). Perhaps what this suggests is that we need to gain a deeper mathematical understanding of bootstrapping in non-linear settings (Brandfonbrener and Bruna, 2020), long-term temporal information transfer (Sutton et al., 1999b), and more generally of the dynamics of self-supervision (Hadsell et al., 2006; Chen et al., 2020). With novel insights, outperforming naive supervised learning tools may become easy.

5

Generative Models through Bootstrapping

This chapter synthesizes my contribution to a novel method to train generative models of discrete objects (Bengio et al., 2021).

In the previous chapters we've analyzed the bootstrapping mechanism as it is used in the literature, through the TD mechanism and its variants. Here we build a novel method around the realization that bootstrapping can be used to estimate **flow** in a directed graph, like water in pipes. In turn, this flow estimation can be used to train generative models that have a particular feature: they generate objects based on a continuous scalar signal rather than the traditional set of positive and negative examples used for many generative models. In that sense, these generative models are closer to RL agents.

This diversity feature is practical in many settings in which exploration is beneficial. Of particular interest here, we will discuss exploration in the biochemical space of small drug-like molecules. In this domain, we show that the proposed method shines by its capacity to discover a large number of chemically interesting molecules.

In the next sections, we closely follow the original material of our contribution (Bengio et al., 2021), but also include additional explanations, results, and insights into future work. I note that this contribution was born out of a larger project focused on drug discovery, and includes two additional aspects, active learning and in-silico biochemistry, as well as proofs for several interesting properties of the proposed method. All the credit for those parts of the work go to my coauthors, and as such discussion on these topics is kept to a minimum. All implementations of the described methods are available at https://github.com/bengioe/gflownet.

5.1 Generating Diverse Rewards

The maximization of expected return R in RL is generally achieved by putting all the probability mass of the policy π on the highest-return sequence of actions. Here, we study the scenario where our objective is not to generate the single highest-reward sequence of actions but rather to sample a distribution of trajectories whose probability is proportional to a given positive return or reward function. This can be useful in tasks where exploration

is important, i.e., we want to sample from the leading modes of the return function. This is equivalent to the problem of turning an energy function into a corresponding generative model, where the object to be generated is obtained via a sequence of actions. By changing the temperature of the energy function (i.e., scaling it multiplicatively) or by taking the power of the return, one can control how selective the generator should be, i.e., only generate from around the highest modes at low temperature or explore more with a higher temperature.

A motivating application for this setup is iterative black-box optimization where the learner has access to an oracle which can compute a reward for a large batch of candidates at each round, e.g., in drug-discovery applications. Diversity of the generated candidates is particularly important when the oracle is itself uncertain. For example, it may consist of cellular assays which is a cheap proxy for clinical trials, or it may consist of the result of a docking simulation (estimating how well a candidate small molecule binds to a target protein) which is a proxy for more accurate but more expensive downstream evaluations (like cellular assays or in-vivo assays in mice).

When calling the oracle is expensive (e.g. it involves a biological experiment), a standard way (Angermueller et al., 2020) to apply machine learning in such exploration settings is to take the data already collected from the oracle (say a set of (x, y) pairs where x is a candidate solution an y is a scalar evaluation of x from the oracle) and train a supervised proxy f (viewed as a simulator) which predicts y from x. The function f or a variant of f which incorporates uncertainty about its value, like in Bayesian optimization (Srinivas et al., 2010; Negoescu et al., 2011), can then be used as a reward function R to train a generative model or a policy that will produce a batch of candidates for the next experimental assays.

In this setup, searching for x which maximizes $R(x)$ is not sufficient because we would like to sample a representative set of x's with high values of R, i.e., around modes of $R(x)$. Note that alternative ways to obtain diversity exist, e.g., with batch Bayesian optimization (Kirsch et al., 2019). An advantage of the proposed approach is that the computational cost is linear in the size of the batch (by opposition with methods which compare pairs of candidates, which is at least quadratic). With the possibility of assays of a hundred thousand candidates using synthetic biology, linear scaling would be a great advantage.

In this chapter, we thus focus on the specific machine learning problem of turning a given positive reward or return function into a generative policy which samples with a probability proportional to the return. Formally, we want to train models that generate x such that:

$$p(x) \propto R(x)^{\beta} \tag{5.1}$$

where $R(x) > 0$ and $\beta \geq 1$ (we ignore β for much of the rest of this discussion as it can be folded into R).

This would be useful to sample novel drug-like molecules when given a reward function R that scores molecules based on their chemical properties. Being able to sample from the high modes of $R(x)$ would provide diversity in the batches of generated molecules sent to assays. This is in contrast with the typical RL objective of maximizing return which we have found to often end up focusing around one or very few good molecules. In our context, $R(x)$ is a proxy for the actual values obtained from assays, which means it can be called often and cheaply, and can be implemented as another DNN. $R(x)$ is retrained or fine-tuned each time we acquire new data from the assays.

To achieve this, we use the notion of flow networks, which we describe in the next section, §5.2, after having briefly surveyed related literature.

5.1.1 A Primer on MCMC

In this chapter we compare our method to Monte Carlo Markov Chain (MCMC) methods, in particular to MCMC methods using the so-called Metropolis-Hastings algorithm (Metropolis et al., 1953; Hastings, 1970). This algorithm is designed to sample unnormalized probability distributions, that is, sample some x with probability proportional to some $f(x) > 0$, exactly as in our desiderata in Eq. 5.1.

Metropolis-Hastings MCMC works as follows: given a proposal distribution $g(x'|x)$, iteratively generate a sequence of x_t, a Markov Chain, such that

$$x_{t+1} = \begin{cases} x' & \text{with probability } \min\left\{1, \dfrac{f(x')}{f(x_t)} \dfrac{g(x_t|x')}{g(x'|x_t)}\right\} \\ x_t & \text{otherwise} \end{cases}$$
$$\text{with } x' \sim g(\cdot|x_t)$$

After some number of steps T, we are guaranteed to have sampled $x_T \sim p(x)$. In other words, after T steps x_T is independent from any choice of x_0 with high probability. This is known as the burn-in period.

While a fundamental method, MCMC has a number of drawbacks (which the method proposed in this chapter aims to address); for a reference see Robert and Casella (2004) Ch. 7. One common problem is that for many distributions the burn-in time may be arbitrarily large; samples may never become truly independent from x_0 in a reasonable amount of computation time. This is often due to *mode separation*: imagine walking in a city located on an island with only one bridge to the mainland, taking a *random* turn at every intersection. The probability of randomly taking the right series of turns that leads

out of the city is extremely low, and vice versa. We would say that the mainland and the island are well separated modes, as it would take a very long time for a Markov Chain to switch from one mode to the other.

Another problem, even supposing T is reasonable, is that sampling n times from p with MCMC has complexity $O(nT)$. To get around this, a common solution is to subsample from the Markov Chain, i.e. choose to produce a sample every k step of the chain. While this strategy is adopted by modern MCMC applications (Xie et al., 2021), it produces *correlated* samples, meaning that the final set of n samples produced may have different a empirical distribution than the intended $p(x) \propto f(x)$.

5.1.2 Literature Review

The objective of training a policy generating states with a probability proportional to rewards was presented by Buesing et al. (2019) but the proposed method only makes sense when there is a bijection between action sequences and states. In contrast, our method is applicable in the more general setting where many paths can lead to the same state. The objective to sample with probability proportional to a given unnormalized positive function is achieved by many MCMC methods (Grathwohl et al., 2021; Dai et al., 2020). However, when mixing between modes is challenging (e.g., in high-dimensional spaces with well-separated modes occupying a tiny fraction of the total volume) convergence to the target distribution can be extremely slow. In contrast, our method is not iterative and amortizes the challenge of sampling from such modes through a training procedure which must be sufficiently exploratory.

This sampling problem comes up in molecule generation and has been studied in this context with numerous generative models (Shi et al., 2020; Luo et al., 2021; Jin et al., 2020), MCMC methods (Seff et al., 2019; Xie et al., 2021), RL (Gottipati et al., 2020; Popova et al., 2019; Cao and Kipf, 2018) and evolutionary methods (Brown et al., 2004; Jensen, 2019; Swersky et al., 2020). Some of these methods rely on a given set of "positive examples" (high-reward) to train a generative model, thus not taking advantage of the "negative examples" and the continuous nature of the measurements (some examples should be generated more often than others). Others rely on the traditional return maximization objectives of RL, which tends to focus on one or a few dominant modes, as we find in our experiments.

The objective that we later formulate in (5.16) may remind the reader of the objective of control-as-inference's Soft Q-Learning (Haarnoja et al., 2017), with the difference that we include *all* the parents of a state in the in-flow, whereas Soft Q-Learning only uses the parent contained in the trajectory. This induces a different policy, as we later show

in Proposition 1, one where $P(\tau) \propto R(\tau)$ rather than $P(x) \propto R(x)$. More generally, we only consider deterministic generative settings whereas RL is a more general framework for stochastic environments. While we believe this framework can be adapted to broader settings, this particular setting offers simplifications that allow us to create efficient generative models for an entire class of problems.

Literature at the intersection of network flow and deep learning is sparse, and is mostly concerned with solving maximum flow problems (Nazemi and Omidi, 2012; Chen and Zhang, 2020) or classification within existing flow networks (Rahul et al., 2017; Pektaş and Acarman, 2019).

5.2 Flow Networks

We now introduce the central concept of this chapter, **flow networks**.

5.2.1 Setting Description

Consider a discrete set $\mathcal{X}$ and policy $\pi(a|s)$ to sequentially construct an object $x \in \mathcal{X}$ with probability $u(x)$ with

$$u(x) \approx \frac{R(x)}{Z} = \frac{R(x)}{\sum_{x' \in \mathcal{X}} R(x')} \tag{5.2}$$

where $R(x) > 0$ is a reward for object x, and $Z = \sum_{x' \in \mathcal{X}} R(x')$ is the so-called *partition function*.

A bit of setup: we are in an MDP setting. Let $\mathcal{S}$ denote the set of states and $\mathcal{X} \subset S$ denote the set of terminal states. There is a unique initial state s_0. Let $\mathcal{A}$ be a finite set, the alphabet, $\mathcal{A}(s) \subseteq \mathcal{A}$ be the set of allowed actions at state s, and let $\mathcal{A}^*(s)$ be the set of all sequences of actions allowed after state s. To every action sequence $\vec{a} = (a_1, a_2, a_3, ..., a_h)$ of $a_i \in \mathcal{A}, h \leq H$ corresponds a single x, i.e. the environment is deterministic so we can define a function F mapping a sequence of actions $\vec{a}$ to an x. If such a sequence is 'incomplete' we define its reward to be 0, i.e. only terminal states have reward and this reward is always positive.

What method should one use to generate batches sampled from $\pi(x) \propto R(x)$? Let's first think of the state space under which we would operate.

When the correspondence between action sequences and states is **bijective**, a state s is uniquely described by some sequence $\vec{a}$, and we can visualize the generative process as the traversal of a tree from a single root node to a leaf corresponding to the sequence of actions along the way. In particular, the TreeSample method of Buesing et al. (2019) can be seen as a special case of the method we propose, i.e., allocating to each node s a value

corresponding to the sum of all the rewards $R(x)$ over the terminal states or leaves of the subtree rooted at s.

However, when this correspondence is **non-injective**, i.e. when multiple action sequences describe the same x, things get trickier. Instead of a tree, we get a directed acyclic graph or DAG (assuming that the sequences must be of finite length, i.e., there are no deterministic cycles), as illustrated in Figure 5.1. For example, and of interest here, molecules can be seen as graphs, which can be described in multiple orders (canonical representations such as SMILES strings also have this problem: there may be multiple descriptions for the same actual molecule).

The standard approach to such a sampling problem is to use iterative MCMC methods (Xie et al., 2021; Grathwohl et al., 2021). Another option is to relax the desire to have $p(x) \propto R(x)$ and to use non-iterative (sequential) RL methods (Gottipati et al., 2020), but these are at high risk of getting stuck in local maxima and of missing modes. Indeed, in our setting, the policy which maximizes the expected return (which is the expected final reward) generates the sequence with the highest return (i.e., a single molecule rather than a set of them, which is what we desire).

In what follows we propose the Generative Flow Network framework, or GFlowNet, which enables us to learn policies such that $p(x) \propto R(x)$ when sampled. We first discuss why existing methods are inadequate, and then show how we can use the metaphor of flows, sinks and sources, to construct adequate policies. We then show that such policies can be learned via a flow-matching objective.

Problems with standard methods and the non-injective case In the bijective case, one can think of the sequential generation of one x as an episode in a tree-structured deterministic MDP, where all leaves x are terminal states (with reward $R(x)$) and the root is the unique initial state s_0. Interestingly, in such a case one can express the pseudo-value of a state $\tilde{V}(s)$ as the sum of all the rewards of the descendants of s (Buesing et al., 2019).

In the non-injective case, these methods are inadequate. Constructing $u(\tau) \approx R(\tau)/Z$, e.g. as per Buesing et al. (2019), MaxEnt RL (Haarnoja et al., 2017), or via an autoregressive method (Nash and Durkan, 2019; Shi et al., 2021) has a particular problem as shown below: if multiple action sequences $\vec{a}$ (i.e. multiple trajectories τ) lead to a final state x, then a serious bias can be introduced in the generative probabilities. Let us denote $\vec{a}+\vec{b}$ as the concatenation of the two sequences of actions $\vec{a}$ and $\vec{b}$, and by extension $s+\vec{b}$ the state reached by applying the actions in $\vec{b}$ from state s.

Proposition 1. *Let $C : \mathcal{A}^* \mapsto \mathcal{S}$ associate each allowed action sequence $\vec{a} \in \mathcal{A}^*$ to a state $s = C(\vec{a}) \in \mathcal{S}$. Let $\tilde{V} : \mathcal{S} \mapsto \mathbb{R}^+$ associate each state $s \in \mathcal{S}$ to $\tilde{V}(s) = \sum_{\vec{b}\in\mathcal{A}^*(s)} R(s+\vec{b}) > 0$,*

where $\mathcal{A}^*(s)$ *is the set of allowed continuations from* s *and* $s+\vec{b}$ *denotes the resulting state, i.e.,* $\tilde{V}(s)$ *is the sum of the rewards of all the states reachable from* s*. Consider a policy* π *which starts from the state corresponding to the empty string* $s_0 = C(\emptyset)$ *and chooses from state* $s \in \mathcal{S}$ *an allowable action* $a \in \mathcal{A}(s)$ *with probability* $\pi(a|s) = \frac{\tilde{V}(s+a)}{\sum_{b\in\mathcal{A}(s)} \tilde{V}(s+b)}$*. Denote* $u(\vec{a} = (a_1, \ldots, a_N)) = \prod_{i=1}^{N} \pi(a_i|C(a_1, \ldots, a_{i-1}))$ *and* $u(s)$ *with* $s \in \mathcal{S}$ *the probability of visiting a state* s *with this policy. The following then obtains:*
(a) $u(s) = \sum_{\vec{a}_i:C(\vec{a}_i)=s} u(\vec{a}_i)$*.*
(b) If C *is bijective, then* $u(s) = \frac{\tilde{V}(s)}{\tilde{V}(s_0)}$ *and as a special case for terminal states* x*,* $u(x) = \frac{R(x)}{\sum_{x\in\mathcal{X}} R(x)}$*.*
(c) If C *is non-injective and there are* $n(x)$ *distinct action sequences* $\vec{a}_i$ *s.t.* $C(\vec{a}_i) = x$*, then* $u(x) = \frac{n(x)R(x)}{\sum_{x'\in\mathcal{X}} n(x')R(x')}$*.*

Proof. Since s can be reached (from s_0) according to any of the action sequences $\vec{a}_i$ such that $C(\vec{a}_i) = s$ and they are mutually exclusive and cover all the possible ways of reaching s, the probability that π visits state s is simply $\sum_{\vec{a}_i:C(\vec{a}_i)=s} u(\vec{a}_i)$, i.e., we obtain (a).
If C is bijective, it means that there is only one such action sequence $\vec{a} = (a_1, \ldots, a_N)$ landing in state s, and the set of action sequences and states forms a tree rooted at s_0. hence by (a) we get that $u(s) = u(\vec{a})$. First note that because $\tilde{V}(s) = \sum_{\vec{b}\in\mathcal{A}^*(s)} R(s + \vec{b})$, i.e., $\tilde{V}(s)$ is the sum of the terminal rewards for all the leaves rooted at s, we have that $\tilde{V}(s) = \sum_{b\in\mathcal{A}(s)} \tilde{V}(s + b)$. Let us now prove by induction that $u(s) = \frac{\tilde{V}(s)}{\tilde{V}(s_0)}$. It is true for $s = s_0$ since $u(s_0 = 1)$ (i.e., every trajectory includes s_0). Assuming it is true for $s' = C(a_1, \ldots, a_{N-1})$, consider $s = C(a_1, \ldots, a_N)$:

$$u(s) = \pi(a_N|s')u(s') = \frac{\tilde{V}(s)}{\sum_{b\in\mathcal{A}(s')} \tilde{V}(s' + b)} \frac{\tilde{V}(s')}{\tilde{V}(s_0)}.$$

Using our above result that $\tilde{V}(s) = \sum_{b\in\mathcal{A}(s)} \tilde{V}(s+b)$, we thus obtain a cancellation of $\tilde{V}(s')$ with $\sum_{b\in\mathcal{A}(s')} \tilde{V}(s' + b)$ and obtain

$$u(s) = \frac{\tilde{V}(s)}{\tilde{V}(s_0)}, \tag{5.3}$$

proving that the recursion holds. We already know from the definition of $\tilde{V}$ that $\tilde{V}(s_0) = \sum_{x\in\mathcal{X}} R(x)$, so for the special case of x a terminal state, $\tilde{V}(x) = R(x)$ and Eq. 5.3 becomes $u(x) = \frac{R(x)}{\sum_{x'\in\mathcal{X}} R(x')}$, which finishes to prove (b).
On the other hand, if C is non-injective, the set of paths forms a DAG, and generally not a tree. Let us transform the DAG into a tree by creating a new state-space (for the tree version) which is the action sequence itself. Note how the same original leaf node x is now repeated $n(x)$ times in the tree (with leaves denoted by action sequences $\vec{a}_i$) if there are

$n(x)$ action sequences leading to x in the DAG. With the same definition of $\tilde{V}$ and $\pi(a|s)$ but in the tree, we obtain all the results from (b) (which are applicable because we have a tree), and in particular $u(\vec{a}_i)$ under the tree is proportional to $R(x') = R(x)$. Applying (a), we see that $u(x) \propto n(x)R(x)$, which proves (c). □

In combinatorial spaces, such as for molecules, where C is non-injective (there are many ways to construct a molecule), this can become exponentially bad as trajectory lengths increase. It means that larger molecules would be exponentially more likely to be sampled than smaller ones, just because of the many more paths leading to them. This creates an imbalance when sampling the model which can hurt exploration, learning, and eventually sampling, if we wish to use the learned model as a true generative model with the property $p(x) \propto R(x)$.

5.2.2 Flow Networks as Generative Models

In the previous scenario, the pseudo-value $\tilde{V}$ is "misinterpreting" the MDP's structure as a tree, leading to the wrong $\pi(x)$. An alternative is to see the MDP as a **flow network**, that is, leverage the DAG structure of the MDP, and learn a flow F, rather than estimating the pseudo-value $\tilde{V}$ as a sum of descendant rewards, as elaborated below.

For what follows, it's useful to think of the flow network as a set of pipes, and of the flow itself as the amount of liquid that goes through each pipe. The pipes, the edges, connect nodes to each other. Nodes are either sources (they make liquid come in from the outside), intermediaries (all the liquid coming into them goes out to other nodes), or sinks (all their liquid is thrown out into the void).

For our use, we define the flow network as a having a single source, the root node (or initial state) s_0 with in-flow Z, and one sink for each leaf (or terminal state) x with out-flow $R(x) > 0$. We write $T(s,a) = s'$ to denote that the state-action pair (s,a) leads to state s'. Note that because C is not a bijection, i.e., there are many paths (action sequences) leading to some node, a node can have multiple parents, i.e. $|\{(s,a) \mid T(s,a) = s'\}| \geq 1$, except for the root, which has no parent. We write $F(s,a)$ for the flow between node s and node $s' = T(s,a)$, $F(s)$ for the total flow going through s.[1] This construction is illustrated in Fig. 5.1.

To satisfy flow conditions, we require that for any node, the incoming flow equals the outgoing flow, which is the total flow $F(s)$ of node s. Boundary conditions are given by

[1]In some sense, $F(s)$ and $F(s,a)$ are close to V and Q, RL's value and action-value functions. These effectively inform an agent taking decisions at each step of an MDP to act in a desired way. With some work, we can also show an equivalence between $F(s,a)$ and the "real" $Q^{\hat{\pi}}$ of some policy $\hat{\pi}$ in a modified MDP (see 5.2.3.2).

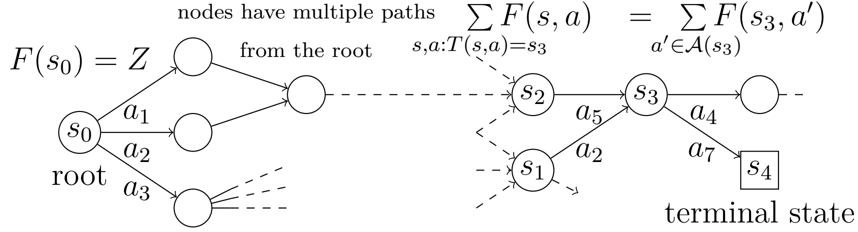


Figure 5.1: A flow network MDP. Episodes start at source s_0 with flow Z. There are no cycles. Terminal states are sinks with out-flow $R(s)$. Exemplar state s_3 has parents $\{(s,a)|T(s,a) = s_3\} = \{(s_1,a_2),(s_2,a_5)\}$ and allowed actions $\mathcal{A}(s_3) = \{a_4,a_7\}$. s_4 is a terminal sink state with $R(s_4) > 0$ and only one parent. The goal is to estimate $F(s,a)$ such that the flow equations are satisfied for all states: for each node, incoming flow equals outgoing flow.

the flow into the terminal nodes x, $R(x)$. Formally, for any node s', we must have that the in-flow

$$F(s') = \sum_{s,a:T(s,a)=s'} F(s,a) \tag{5.4}$$

equals the out-flow

$$F(s') = \sum_{a'\in\mathcal{A}(s')} F(s',a'). \tag{5.5}$$

More concisely, with $R(s') = 0$ for interior nodes and $\mathcal{A}(s') = \varnothing$ for leaf (sink/terminal) nodes, we can write:

$$\sum_{s,a:T(s,a)=s'} F(s,a) = R(s') + \sum_{a'\in\mathcal{A}(s')} F(s',a'), \tag{5.6}$$

where $F(s,a)$ is the edge flow, $F(s,a) > 0\ \forall s,a$ (for this we needed to constrain $R(x)$ to be positive too). One could include in principle nodes and edges with zero flow but it would make it difficult to talk about the logarithm of the flow, as we do below, and such states can always be excluded by the allowed set of actions for their parent states.

We show that such a flow correctly produces $u(x) = R(x)/Z$ when the above three flow equations are satisfied.

Proposition 2. *Let us define a policy π that generates trajectories starting in state s_0 by sampling actions $a \in \mathcal{A}(s)$ according to*

$$\pi(a|s) = \frac{F(s,a)}{F(s)} = \frac{F(s,a)}{\sum_{a'} F(s,a')} \tag{5.7}$$

where $F(s,a) > 0$ is the flow through allowed edge (s,a), $F(s) = R(s) + \sum_{a\in\mathcal{A}(s)} F(s,a)$ where $R(s) = 0$ for non-terminal nodes s and $F(x) = R(x) > 0$ for terminal nodes x, and the flow equation $\sum_{s,a:T(s,a)=s'} F(s,a) = R(s') + \sum_{a'\in\mathcal{A}(s')} F(s',a')$ is satisfied. Let $u(s)$

denote the probability of visiting state s when starting at s_0 and following $\pi(\cdot|\cdot)$. Then
(a) $u(s) = \frac{F(s)}{F(s_0)}$
(b) $F(s_0) = \sum_{x \in \mathcal{X}} R(x)$
(c) $u(x) = \frac{R(x)}{\sum_{x' \in \mathcal{X}} R(x')}$.

Proof. We have $u(s_0) = 1$ since we always start in root node s_0. Note that $\sum_{x \in \mathcal{X}} u(x) = 1$ because terminal states are mutually exclusive, but in the case of non-bijective C, we cannot say that $\sum_{s \in \mathcal{S}} u(s)$ equals 1 because the different states are not mutually exclusive in general. Then

$$u(s') = \sum_{(a,s):T(s,a)=s'} \pi(a|s)u(s) \tag{5.8}$$

i.e., using Eq. 5.7,

$$u(s') = \sum_{(a,s):T(s,a)=s'} \frac{F(s,a)}{F(s)} u(s). \tag{5.9}$$

We can now conjecture that the statement

$$u(s) = \frac{F(s)}{F(s_0)} \tag{5.10}$$

is true and prove it by induction. This is trivially true for the root, which is our base statement, since $u(s_0) = 1$. By induction, we then have that if the statement is true for parents s of s', then

$$u(s') = \sum_{s,a:T(s,a)=s'} \frac{F(s,a)}{F(s)} \frac{F(s)}{F(s_0)} = \frac{\sum_{s,a:T(s,a)=s'} F(s,a)}{F(s_0)} = \frac{F(s')}{F(s_0)} \tag{5.11}$$

which proves the statement, i.e., the first conclusion (a) of the theorem. We can then apply it to the case of terminal states x, whose flow is fixed to $F(x) = R(x)$ and obtain

$$u(x) = \frac{R(x)}{F(s_0)}. \tag{5.12}$$

Noting that $\sum_{x \in X} u(x) = 1$ and summing both sides of Eq. 5.12 over x we thus obtain (b), i.e., $F(s_0) = \sum_{x \in \mathcal{X}} R(x)$. Plugging this back into Eq. 5.12, we obtain (c), i.e., $u(x) = \frac{R(x)}{\sum_{x' \in \mathcal{X}} R(x')}$. □

Thus our choice of π satisfies our desiderata: it maps a reward function R to a generative model which generates x with probability $u(x) \propto R(x)$, whether C is bijective or non-injective (the former being a special case of the latter, and we just provided a proof for the general non-injective case–in other words, trees are DAGs but not vice versa, and we proved this for DAGs).

5.2.3 Objective Functions for Flow Matching

We can now leverage our RL intuitions to create a learning algorithm out of the above theoretical results. In particular, we propose to approximate the flows F such that the flow conditions are obtained at convergence with enough capacity in our estimator of F, just like the Bellman conditions for temporal-difference (TD) algorithms (Sutton and Barto, 2018).

Recall that we've defined the flow network such that for any node s', we have the following equalities, the first for the in-flow

$$F(s') = \sum_{s,a:T(s,a)=s'} F(s,a), \tag{5.13}$$

and the second for the out-flow

$$F(s') = \sum_{a' \in \mathcal{A}(s')} F(s',a'). \tag{5.14}$$

By approximating F and matching these equalities, i.e. by a kind of bootstrapping mechanism, we recover the policy as in (5.7) and obtain a generative model. This could yield the following objective for a trajectory τ:

$$\tilde{\mathcal{L}}_\theta(\tau) = \sum_{s' \in \tau \neq s_0} \left(\sum_{s,a:T(s,a)=s'} F_\theta(s,a) - R(s') - \sum_{a' \in \mathcal{A}(s')} F_\theta(s',a') \right)^2 . \tag{5.15}$$

One issue from a learning point of view is that the flow will be very large for nodes near the root (early in the trajectory) and tiny for nodes near the leaves (late in the trajectory). In high-dimensional spaces where the cardinality of $\mathcal{X}$ is exponential (e.g., in the typical number of actions to form an x), $F(s,a)$ for early states will be exponentially larger than for later states. Since we want $F(s,a)$ to be the output of a neural network, this would lead to serious numerical issues.

To avoid this problem, we define the flow matching objective on a log-scale, where we match not the incoming and outgoing flows but their logarithms, and we train our predictor to estimate $F^{\log}_\theta(s,a) = \log F(s,a)$, and exponentiate-sum-log the $F^{\log}_\theta$ predictions to compute the loss, yielding the square of a difference of logs:

$$\mathcal{L}_{\theta,\epsilon}(\tau) = \sum_{s' \in \tau \neq s_0} \left(\log \left[\epsilon + \sum_{s,a:T(s,a)=s'} \exp F^{\log}_\theta(s,a) \right] - \log \left[\epsilon + R(s') + \sum_{a' \in \mathcal{A}(s')} \exp F^{\log}_\theta(s',a') \right] \right)^2 \tag{5.16}$$

which gives equal gradient weighing to large and small magnitude predictions. Note that matching the logs of the flows is equivalent to making the ratio of the incoming and outgoing flow closer to 1. To give more weight to errors on large flows and avoid taking

the logarithm of a tiny number, we compare log(ϵ+incoming flow) with log(ϵ+outgoing flow). It does not change the global minimum, which is still when the flow equations are satisfied, but it avoids numerical issues with taking the log of a tiny flow.

The hyper-parameter ϵ trades-off how much pressure we put on matching large versus small flows. Since we want to discover the top modes of R, it makes sense to care more for the larger flows. In practice, we find that using ϵ close to the minimal effective reward in the environment makes learning stable.

Many other objectives are possible for which flow matching is also a global minimum. We will come back to this later in the chapter.

In what follows we use F_θ to denote the parameterized flow predictor, and $u_\theta(x)$ for the distribution on $\mathcal{X}$ induced by F_θ.

We call this method to train generative models using flow networks **GFlowNet**.

5.2.3.1 GFlowNet is an off-policy method

An interesting advantage of such objective functions is that they yield off-policy offline methods. The predicted flows F do not depend on the policy used to sample trajectories (apart from the fact that the samples should sufficiently cover the space of trajectories in order to obtain generalization). This is formalized below, which shows that we can use any broad-support policy to sample training trajectories and still obtain the correct flows and generative model, i.e., training can be off-policy.

Proposition 3. *Let trajectories τ used to train F_θ be sampled from an exploratory policy P with the same support as the optimal π defined in Eq. 5.7 for a correct flow F^*. Also assume that $\exists\theta : F_\theta = F^*$, i.e., we choose a sufficiently rich family of predictors. Let $\theta^* \in \text{argmin}_\theta E_{P(\tau)}[L_\theta(\tau)]$ a minimizer of the expected training loss. Let $L_\theta(\tau)$ have the property that when flows are matched it achieves its lowest possible value, 0. First, it can be shown that this property is satisfied for the loss in Eq. 5.16. Then*

$$F_{\theta^*} = F^*, \quad \text{and} \tag{5.17}$$

$$L_{\theta^*}(\tau) = 0 \quad \forall\tau \sim P(\tau), \tag{5.18}$$

i.e., a global optimum of the expected loss provides the correct flows. If

$$\pi_{\theta^*}(a|s) = \frac{F_{\theta^*}(s,a)}{\sum_{a'\in\mathcal{A}(s)} F_{\theta^*}(s,a')}$$

then we also have

$$\pi_{\theta^*}(x) = \frac{R(x)}{Z}. \tag{5.19}$$

Proof. A per-trajectory loss of 0 can be achieved by choosing a θ such that $F_\theta = F^*$ (which we assumed was possible), since this makes the incoming flow equal the outgoing flow. Note that there always exists a solution F^* in the space of allow possible flow functions which satisfies the flow equations (incoming = outgoing) by construction of flow networks with only a constraint on the flow in the terminal nodes (leaves).

Since having $L_\theta(\tau)$ equal to 0 for all $\tau \sim P(\tau)$ makes the expected loss 0, and this is the lowest achievable value (since $L_\theta(\tau) \geq 0 \;\; \forall\theta$), it means that such a θ is a global minimizer of the expected loss, and we can denote it θ^*.

Since we have chosen P with support large enough to include all the trajectories leading to a terminal state $R(x) > 0$, it means that $L_\theta(\tau) = 0$ for all these trajectories and that $F_\theta = F^*$ for all nodes on these trajectories. We can then apply Proposition 2 (since the flows match everywhere and we have defined the policy correspondingly, as per Eq. 5.7). We then obtain the conclusion by applying result (c) from Proposition 2. □

5.2.3.2 Action-value function equivalence

Here we show that the flow $F(s,a)$ that the proposed method learns can correspond to a "real" action-value function $\hat{Q}^\mu(s,a)$ in an RL sense, for a policy μ.

First note that this is in a way trivially true: in inverse RL (Ng et al., 2000) there typically exists an infinite number of solutions to defining $\hat{R}$ from a policy π such that $\pi = \arg\max_{\pi_i} F^{\pi_i}(s;\hat{R}) \; \forall s$, where we will adapt our notation to indicate that $F^{\pi_i}(s;\hat{R})$ is the flow function at s induced by solving the flow-matching problem for some reward function $\hat{R}$. We apply the same notation for Q.

More interesting is the case where $F(s,a;R)$ obtained from computing the flow corresponding to R is exactly equal to some $Q^\mu(s,a;\hat{R})$ modulo a multiplicative factor $f(s)$. What are μ and $\hat{R}$? In the bijective case a simple answer exists.

Proposition 4. *Let μ be the uniform policy such that $\mu(a|s) = 1/|\mathcal{A}(s)|$, let $f(x) = \prod_{t=0}^{n} |\mathcal{A}(s_t)|$ when $x \equiv (s_0, s_1, ..., s_n)$, and let $\hat{R}(x) = R(x)f(s_{n-1})$, then $Q^\mu(s,a;\hat{R}) = F(s,a;R)f(s)$.*

Proof. By definition of the action-value function in terms of the action-value at the next step and by definition of μ:

$$Q^\mu(s,a;\hat{R}) = \hat{R}(s') + \frac{1}{|\mathcal{A}(s')|} \sum_{a' \in \mathcal{A}(s')} Q^\mu(s',a';\hat{R}) \tag{5.20}$$

where $s' = T(s,a)$, as the environment is deterministic and has a tree structure.

For some leaf s', $Q^\mu(s,a;\hat{R}) = \hat{R}(s') = R(s')f(s)$. Again for some leaf s', the flow is $F(s,a;R) = R(s')$. Thus $Q^\mu(s,a;\hat{R}) = F(s,a;R)f(s)$. Suppose (5.20) is true, then by

induction for a non-leaf s':

$$Q^{\mu}(s,a;\hat{R}) = \hat{R}(s') + \frac{1}{|\mathcal{A}(s')|}\sum_{a'\in\mathcal{A}(s')} Q^{\mu}(s',a';\hat{R}) \tag{5.21}$$

$$Q^{\mu}(s,a;\hat{R}) = 0 + \frac{1}{|\mathcal{A}(s')|}\sum_{a'\in\mathcal{A}(s')} F(s',a';R)f(s') \tag{5.22}$$

we know from (5.6) that

$$F(s,a;R) = \sum_{a'\in\mathcal{A}(s')} F(s',a';R) \tag{5.23}$$

and since $f(s') = f(s)|\mathcal{A}(s')|$, we have that:

$$Q^{\mu}(s,a;\hat{R}) = \frac{F(s,a;R)f(s')}{|\mathcal{A}(s')|} \tag{5.24}$$

$$= \frac{F(s,a;R)f(s)|\mathcal{A}(s')|}{|\mathcal{A}(s')|} \tag{5.25}$$

$$= F(s,a;R)f(s) \tag{5.26}$$

□

Thus we have shown that the flow in a bijective case corresponds to the action-value of the uniform policy. This result suggests that the policy evaluation of the uniform policy learns something non-trivial in the tree MDP case. Perhaps such a quantity could be used in other interesting ways.

In the non-injective case, since an infinite number of valid flows exists, it's not clear that such a simple equivalence always exists.

As a particular case, consider the flow F which assigns exactly 0 flow to edges that would induce multiple paths to any node. In other words, consider the flow which induces a tree, i.e. a bijection between action sequences and states, by disallowing flow between edges not in that bijection. By Proposition 4, we can recover some valid Q^{μ}.

Since there is at least one flow for which this equivalence exists, we conjecture that more general mappings between flows and action-value functions exist.

Conjecture *There exists f a function of $n(s)$ the number of paths to s, $\mathcal{A}(s)$, and $n_p(s) = |\{(p,a)|T(p,a) = s\}|$ the number of parents of s, such that*

$$f(s, n(s), n_p(s), \mathcal{A}(s))Q^{\mu}(s,a;\hat{R}) = F(s,a;R)$$

and $\hat{R}(x) = R(x)f(x)$ for the uniform policy μ and for some valid flow $F(s,a;R)$.

5.3 Empirical Results

We first verify that GFlowNet works as advertised on an artificial domain small enough to compute the partition function exactly, and compare its abilities to recover modes compared to standard MCMC and RL methods, with its sampling distribution better matching the normalized reward.

We find that GFlowNet (A) converges to $u(x) \propto R(x)$, (B) requires less samples to achieve some level of performance than MCMC and PPO methods and (C) recovers all the modes and does so faster than MCMC and PPO, both in terms of wall-time and number of states visited and queried.

We then test GFlowNet on a large scale domain, which consists in generating small drug molecule graphs, with a reward that estimates their binding affinity to a target protein (see §5.3.2.1). We find that GFlowNet finds higher reward and more diverse molecules faster than baselines.

All our ML code uses the PyTorch (Paszke et al., 2019a) library. We reimplement RL and other baselines. We use the AutoDock Vina (Trott and Olson, 2010) library for binding energy estimation and RDKit (Landrum) for chemistry routines.

Running all the molecule experiments presented in this chapter takes an estimated 26 GPU days. We use a cluster with NVidia V100 GPUs. The grid experiments take an estimated 8 CPU days (for a single-core).

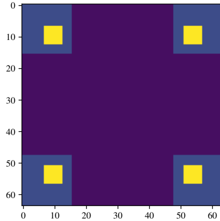
All implementations are available at https://github.com/bengioe/gflownet.

5.3.1 A (hyper-)grid domain

Consider an MDP where states are the cells of a n-dimensional hypercubic grid of side length H. The agent starts at coordinate $(0, 0, ...)$ and is only allowed to increase coordinate i with action a_i (up to H, upon which the episode terminates). A *stop* action indicates to terminate the trajectory. There are many action sequences that lead to the same coordinate, making this MDP a DAG. We associate with each cell of the grid a coordinate in $[-1, 1]^n$, mapping $s = (0, 0, ...)$ to $x = (-1, -1, ..)$. The reward for ending the trajectory in x is some $R(x) > 0$. For MCMC methods, in order to have an ergodic chain, we allow the iteration to decrease coordinates as well, and there is no *stop* action.

We ran experiments with this reward function:

$$R(x) = R_0 + R_1 \prod_i \mathbb{I}(0.5 < |x_i|) + R_2 \prod_i \mathbb{I}(0.6 < |x_i| < 0.8)$$



with $0 < R_0 \ll R_1 < R_2$, pictured above when $n = 2$ and $H = 64$. For this choice of R, there are only interesting rewards near the corners of the grid, and there are exactly 2^n modes. We set $R_1 = 1/2,\ R_2 = 2$. By varying R_0 and setting it closer to 0, we make this problem artificially harder, creating a region of the state space which it is undesirable to explore. To measure the performance of a method, we measure the empirical L1 error $\mathbb{E}[|p(x) - u(x)|]$. $p(x) = R(x)/Z$ is known in this domain, and u is estimated by repeated sampling and counting frequencies for each possible x. We also measure the number of modes with at least 1 visit as a function of the number of states visited.

We run the above experiment for $R_0 \in \{10^{-1}, 10^{-2}, 10^{-3}\}$ with $n = 4$, $H = 8$. In Fig. 5.2 we see that GFlowNet is robust to R_0 and obtains a low L1 error, while a Metropolis-Hastings-MCMC based method requires exponentially more samples than GFlowNet to achieve same levels of L1 error. This is apparent in Fig. 5.2 (with a log-scale horizontal axis) by comparing the slope of progress of GFlowNet (beyond the initial stage) and that of the MCMC sampler. If method 1 has slope m_1 and method 2 has slope m_2, then the ratio of the number of samples needed grows as e^{m_1/m_2}. This validates hypothesis (A).

We also see that MCMC takes much longer to visit each mode *once* as R_0 decreases, while GFlowNet is only slightly affected, with GFlowNet converging to some level of L1 error faster, as per hypothesis (B). This suggests that GFlowNet is robust to the separation between modes (represented by R_0 being smaller) and thus recovers all the modes much faster than MCMC (again, noting the log-scale of the horizontal axis).

To compare to RL, we run PPO (Schulman et al., 2017). To discover all the modes in a reasonable time, we need to set the entropy maximization term much higher (0.5) than usual ($\ll 1$). We verify that PPO is not overly regularized by comparing it to a random agent. PPO finds all the modes faster than uniform sampling, but much more slowly than GFlowNet, and is also robust to the choice of R_0. This and the previous result validates hypothesis (C). We also run SAC (Haarnoja et al., 2018), finding similar or worse results.

Let's now to understand our method, and look at what is learned by GFlowNet. What is the distribution of flows learned?

First, in Figure 5.3 (Left), we can observe that the distribution learned, $u_\theta(x)$, matches almost perfectly $p(x) \propto R(x)$ on a grid where $n = 2$, $H = 8$. In Figure 5.3 (Middle) we

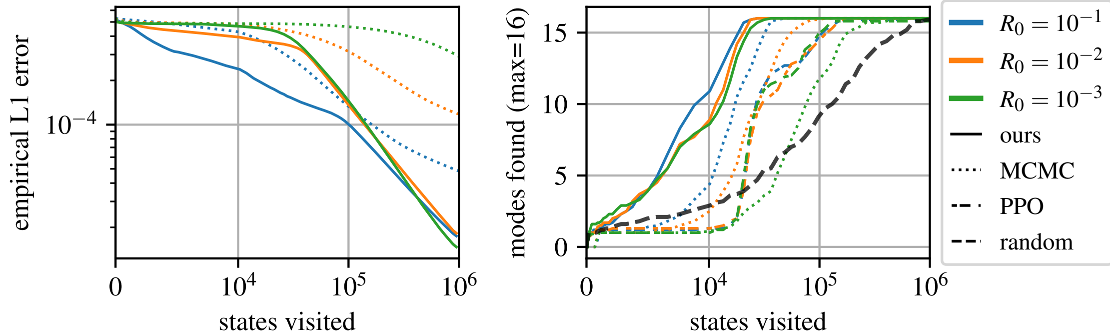


Figure 5.2: Hyper-grid domain. Changing the task difficulty R_0 to illustrate the advantage of GFlowNet over others. We see that as R_0 gets smaller, MCMC struggles to fit the distribution because it struggles to visit all the modes. PPO also struggles to find all the modes, and requires very large entropy regularization, but is robust to the choice of R_0. We plot means over 10 runs for each setting.

plot the visit distribution on all paths that lead to mode $s = (6, 6)$, starting at $s_0 = (0, 0)$. We see that it is fairly spread out, but not uniform: there seems to be some preference towards other corners, presumably due to early bias during learning as well as the position of the other modes. In Figure 5.3 (Right) we plot what the uniform distribution on paths from $(0, 0)$ to $(6, 6)$ would look like for reference. Note that our loss does not enforce any kind of distribution on flows, and a uniform flow is not necessarily desirable (investigating this could be interesting future work, perhaps some distributions of flows have better generalization properties).

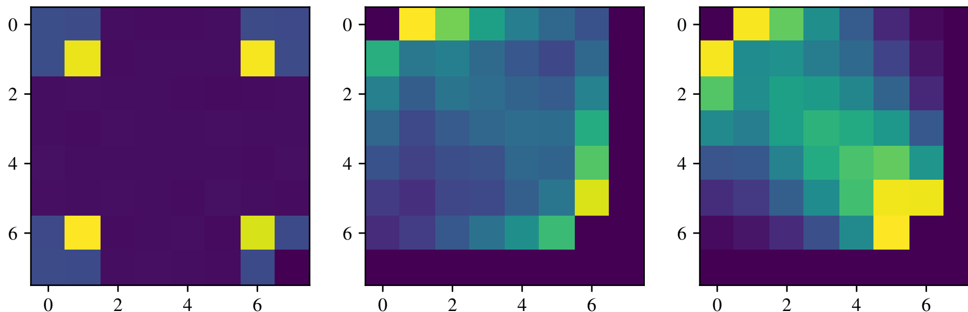


Figure 5.3: Grid with $n = 2$, $H = 8$. Left, the distribution $u_\theta(x)$ learned on the grid matches $p(x)$ almost perfectly; measured by sampling 30k points. Middle, the visit distribution on sampled paths leading to $(6, 6)$. Right, the uniform distribution on all paths leading to $(6, 6)$.

Note that we also ran Soft Actor Critic (Haarnoja et al., 2018) on this domain, but we were unable to find hyperparameters that pushed SAC to find all the modes for $n = 4$, $H = 8$; SAC would find at best 10 of the 16 modes even when strongly regularized (but

not so much so that the policy trivially becomes the uniform policy). While we believe our implementation to be correct, we did not think it would be relevant to include these results in figures, as they are poor but not really surprising: as would be consistent with reward-maximization, SAC quickly finds a mode to latch onto, and concentrates all of its probability mass on that mode, which is the no-diversity failure mode of RL we are trying to avoid with GFlowNet.

Next, let's look at the losses as a function of R_0, back in the $n = 4$, $H = 8$ setting. We separate the loss in two components, the leaf loss (loss for terminal transitions) and the inner flow loss (loss for non-terminals). In Figure 5.4 we see that as R_0 decreases, both inner flow and leaf losses get larger. This is reasonable for two reasons: first, for e.g. with $R_0 = 10^{-3}$, $\log 10^{-3}$ is a larger magnitude number which is harder for DNNs to accurately output, and second, the terminal states for which $\log 10^{-3}$ is the flow output are $100\times$ rarer than in the $R_0 = 10^{-1}$ case (because we are sampling states on-policy), thus a DNN is less inclined to correctly predict their value correctly. This incurs rare but large magnitude losses. Note that theses losses are nonetheless small, in the order of 10^{-3} or less, and at this point the distribution is largely fit and the model is simply converging.

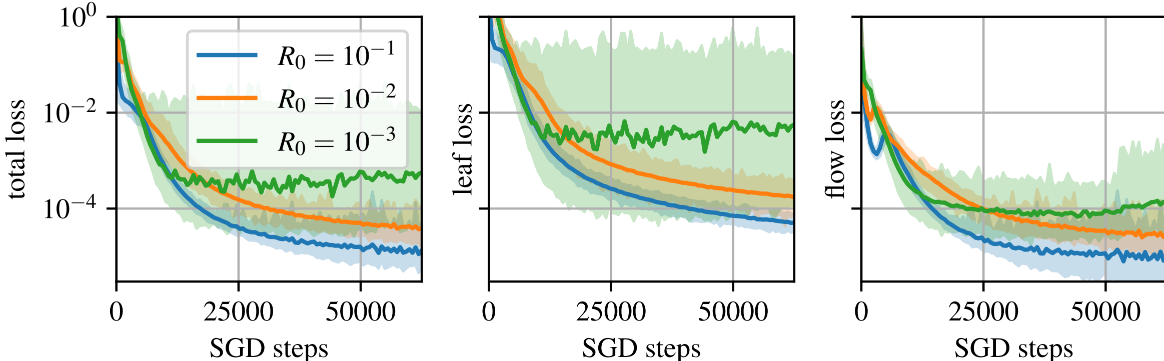


Figure 5.4: Losses during training for the "corners" reward function in the hyper-grid, with $n = 4$, $H = 8$. Shaded regions are the min-max bounds.

GFlowNet as an offline off-policy method To demonstrate this feature of GFlowNet, we train it on a fixed dataset of trajectories and observe what the learned distribution is. For this experiment we use $R(x) = 0.01 + \prod_i(\cos(50x_i) + 1)f_\mathcal{N}(5x_i)$, $f_\mathcal{N}$ is the normal p.d.f., $n = 2$ and $H = 30$. We show results for two random datasets. First, in Figure 5.5 we show what is learned when the dataset is sampled from a uniform random policy, and second in Figure 5.6 when the dataset is created by sampling points uniformly on the grid and walking backwards to the root to generate trajectories. The first setting should be much harder than the second, and indeed the learned distribution matches $p(x)$ much

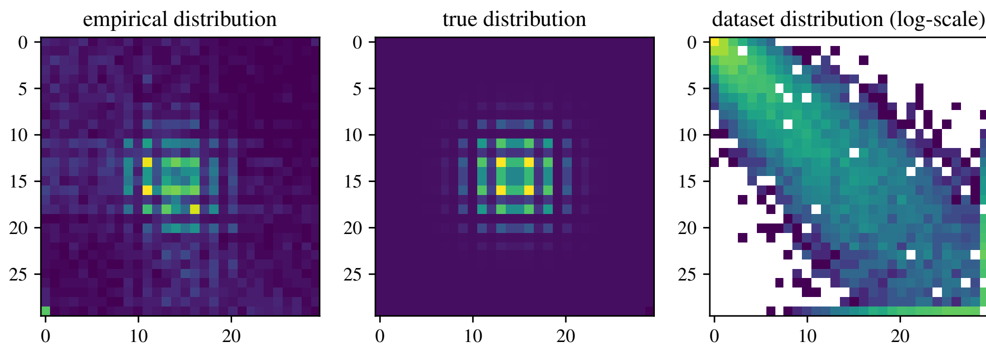


Figure 5.5: Grid with $n = 2$, $H = 30$. Left, the learned distribution $u_\theta(x)$. Middle, the true distribution. Right, the dataset distribution, here generated by executing a uniform random policy from s_0.

better when the dataset points are more uniform. Note that in both cases many points are left out intentionally as a generalization test.

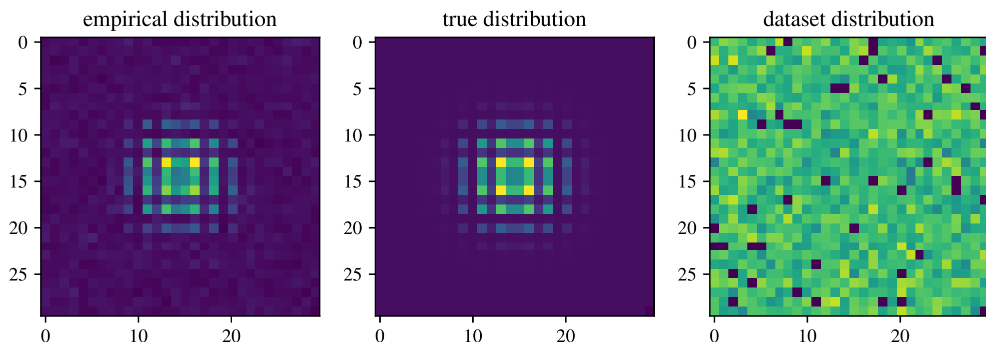


Figure 5.6: Grid with $n = 2$, $H = 30$. Left, the learned distribution $u_\theta(x)$. Middle, the true distribution. Right, the dataset distribution, here generated by sampling a point uniformly on the grid and sampling random parents until s_0 is reached, thus generating a training trajectory in reverse.

These results suggest that GFlowNet can easily be applied offline and off-policy. Note that we did not do hyperparameter search on these two plots, these are purely illustrative and we believe it is likely that better generalization can be achieved by tweaking hyperparameters.

5.3.2 Generating small molecules

Here our goal is to generate a diverse set of small molecules that have a high reward. We define a large-scale environment which allows an agent to sequentially generate molecules.

This environment is challenging, with up to 10^{16} states and between 100 and 2000 actions depending on the state.

We follow the framework of Jin et al. (2020) and generate molecules by parts using a predefined vocabulary of building blocks that can be joined together forming a *junction tree* (detailed in 5.3.2.1). This is also known as fragment-based drug design (Kumar et al., 2012; Xie et al., 2021). Generating such a graph can be described as a sequence of additive edits: given a molecule and constrains of chemical validity, we choose an atom to attach a block to. The action space is thus the product of choosing where to attach a block and choosing which block to attach. There is an extra action to stop the editing sequence. This sequence of edits yields a DAG MDP, as there are multiple action sequences that lead to the same molecule graph, and no edge removal actions, which prevents cycles.

The reward is computed with a pretrained *proxy* model that predicts the binding energy of a molecule to a particular protein target (soluble epoxide hydrolase, sEH, see 5.3.2.1). Although computing binding energy is computationally expensive, we can call this proxy cheaply. Note that for realistic drug design, we would need to consider many more quantities such as drug-likeness (Bickerton et al., 2012), toxicity, or synthesizability. Our goal here is not solve this problem, and our work situates itself within a larger project. Instead, we want to show that given a proxy R in the space of molecules, we can quickly match its induced distribution $u(x) \propto R(x)$ and find many of its modes.

We parameterize the proxy with an MPNN (Gilmer et al., 2017) over the atom graph. Our flow predictor F_θ is parameterized similarly to MARS (Xie et al., 2021), with an MPNN, but over the junction tree graph (the graph of blocks), which had better performance. For fairness, this architecture is used for both GFlowNet and the baselines. Complete details can be found in §5.3.2.2.

We pretrain the proxy with 300k molecules from a 80/20 mix of random trajectories and RL-generated trajectories[2] down to a test MSE of 0.6; molecules are scored according to the docking score, computed with docking (Trott and Olson, 2010), renormalized so that most scores fall between 0 and 10 (to have $R(x) > 0$). We plot the dataset's reward distribution in Fig. 5.8. We train all generative models with up to 10^6 molecules. During training, sampling follows exploratory policy $P(a|s)$ which is a mixture between $\pi(a|s)$ (Eq. 5.7), used with probability 0.95, and a uniform distribution over allowed actions with probability 0.05.

[2]The reasoning for this mix is that populating the dataset with a few more high-scoring molecules (according to the docking oracle) would make for a more interesting generative task.

5.3.2.1 Molecule Domain Chemical Details

We allow the agent to choose from a library of 72 predefined blocks. We duplicate blocks from the point of view of the agent to allow attaching to different symmetry groups of a given block. This yields a total of 105 actions per stem; stems are atoms where new blocks can be attached to. We choose the blocks via the process suggested by Jin et al. (2020) over the ZINC dataset (Sterling and Irwin, 2015). We allow the agent to generate up to 8 blocks in a given molecule.

In chemistry, molecules can be written down using a SMILES notation (Weininger, 1988). These 72 blocks' SMILES are `Br`, `C`, `C#N`, `C1=CCCCC1`, `C1=CNC=CC1`, `C1CC1`, `C1CCCC1`, `C1CCCCC1`, `C1CCNC1`, `C1CCNCC1`, `C1CCOC1`, `C1CCOCC1`, `C1CNCCN1`, `C1COCCN1`, `C1COCC[NH2+]1`, `C=C`, `C=C(C)C`, `C=CC`, `C=N`, `C=O`, `CC`, `CC(C)C`, `CC(C)O`, `CC(N)=O`, `CC=O`, `CCC`, `CCO`, `CN`, `CNC`, `CNC(C)=O`, `CNC=O`, `CO`, `CS`, `C[NH3+]`, `C[SH2+]`, `Cl`, `F`, `FC(F)F`, `I`, `N`, `N=CN`, `NC=O`, `N[SH](=O)=O`, `O`, `O=CNO`, `O=CO`, `O=C[O-]`, `O=PO`, `O=P[O-]`, `O=S=O`, `O=[NH+][O-]`, `O=[PH](O)O`, `O=[PH]([O-])O`, `O=[SH](=O)O`, `O=[SH](=O)[O-]`, `O=c1[nH]cnc2[nH]cnc12`, `O=c1[nH]cnc2c1NCCN2`, `O=c1cc[nH]c(=O)[nH`
`O=c1nc2[nH]c3ccccc3nc-2c(=O)[nH]1`, `O=c1nccc[nH]1`, `S`, `c1cc[nH+]cc1`, `c1cc[nH]c1`, `c1ccc2[nH]ccc2c1`, `c1ccc2ccccc2c1`, `c1ccccc1`, `c1ccncc1`, `c1ccsc1`, `c1cn[nH]c1`, `c1cncnc1`, `c1cscn1`, `c1ncc2nc[nH]c2n1`.

We illustrate these building blocks and their attachment points in Figure 5.7 showing standard planar representations of molecules.

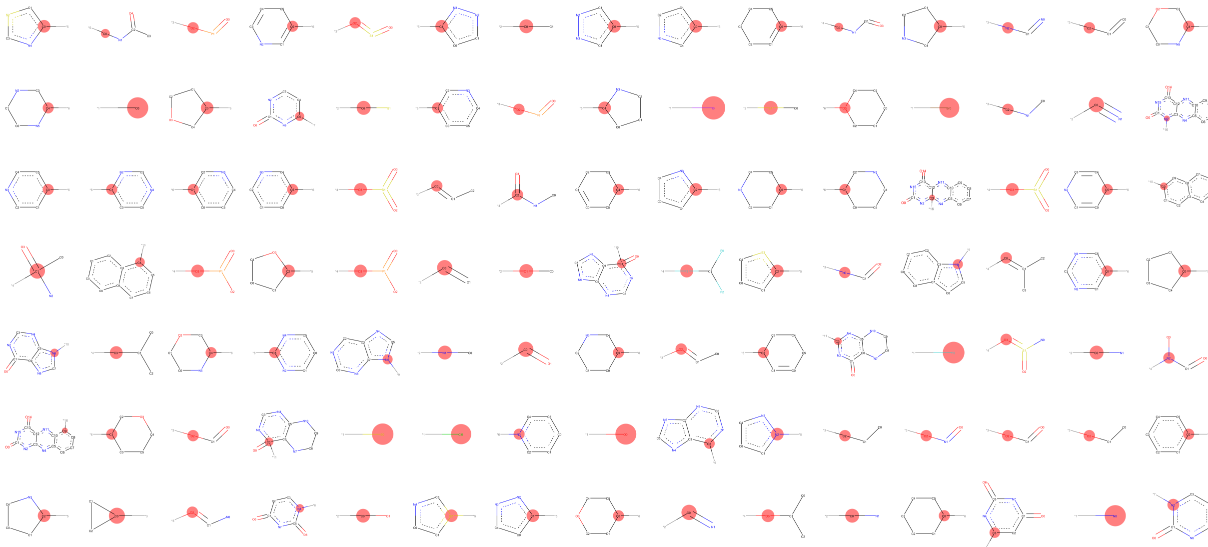


Figure 5.7: The list of building blocks used in molecule design. The stem, the atom which connects the block to the rest of the molecule, is highlighted.

We compute the reward based on a proxy's prediction. This proxy is trained on a dataset of 300k randomly generated molecules, whose binding affinity with a target protein

has been computed with AutoDock (Trott and Olson, 2010). Since the binding affinity is an energy where lower is better, we takes its opposite and then renormalize it (subtract the mean, divide by the standard deviation) to obtain the reward.

We use the sEH protein and its 4JNC inhibitor. The soluble epoxide hydrolase, or sEH, is a well studied protein which plays a role in respiratory and heart disease, which makes it an interesting pharmacological target and benchmark for ML methods.

5.3.2.2 Molecule Domain Architecture Details

For the proxy of the oracle, from which the reward is defined, we use an MPNN (Gilmer et al., 2017) that receives the atom graph as input. We compute the atom graph using RDKit. Each node in the graph has features including the one-hot vector of its atomic number, its hybridization type, its number of implicit hydrogens, and a binary indicator of it being an acceptor or a donor atom. The MPNN uses a GRU at each iteration as the graph convolution layer is applied iteratively for 12 steps, followed by a Set2Set operation to reduce the graph, followed by a 3-layer MLP. We use 64 hidden units in all of its parts, and LeakyReLU activations everywhere (except inside the GRU).

For the flow predictor F we also use an MPNN, but it receives the block graph as input. This graph is a tree by construction. Each node in the graph is a learned embedding (each of the 105 blocks has its own embedding and each type of bond has an edge embedding). We again use a GRU over the convolution layer applied 10 times. For each stem of the graph (which represents an atom or block where the agent can attach a new block) we pass its corresponding embedding (the output of the 10 steps of graph convolution + GRU) into a 3-layer MLP to produce 105 logits representing the probability of attaching each block to this stem for MARS and PPO, or representing the flow $F(s, a)$ for GFlowNet; since each block can have multiple stems, this MLP also receives the underlying atom within the block to which the stem corresponds. For the stop action, we perform a global mean pooling followed by a 3-layer MLP that outputs 1 logit for each flow prediction. We use 256 hidden units everywhere as well as LeakyReLU activations.

For further stability we found that multiplying the loss for terminal transitions by a factor $\lambda_T > 1$ helped. Intuitively, doing so prioritizes correct predictions at the endpoints of the flow, which can then propagate through the rest of the network/state space via our bootstrapping objective. This is similar to using reward prediction as an auxiliary task in RL (Jaderberg et al., 2017).

Here is a summary of the flow model hyperparameters:

Learning rate	5×10^{-4}	
Minibatch size	4	# of trajectories per SGD step
Adam β, ϵ	(0.9, 0.999), 10^{-8}	
# hidden & # embed	256	
# convolution steps	10	
Loss ϵ	2.5×10^{-5}	ϵ in (5.16)
Reward T	8	
Reward β	10	$\hat{R}(x) = (R(x)/T)^{\beta}$
Random action probability	0.05	exploratory factor
λ_T	10	leaf loss coefficient
R_{min}	0.01	R is clipped below R_{min}, i.e. $\hat{R}_{min} = (R_{min}/T)^{\beta}$

For MARS we use a learning rate of 2.5×10^{-4} and for PPO, 1×10^{-4}. For PPO we use an entropy regularization coefficient of 10^{-6} (higher did not help) and we set the reward β to 4 (higher did not help). For MARS we use the same algorithmic hyperparameters as those found in Xie et al. (2021). For JT-VAE, we use the code provided by Jin et al. (2020) as-is, only replacing the reward signal with ours.

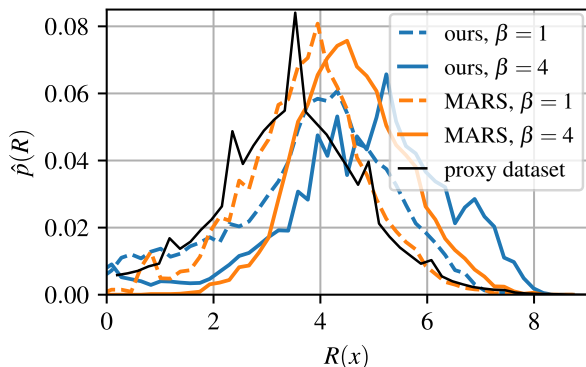


Figure 5.8: Empirical density of rewards. We verify that GFlowNet is consistent by training it with R^{β}, $\beta = 4$, which has the hypothesized effect of shifting the density to the right.

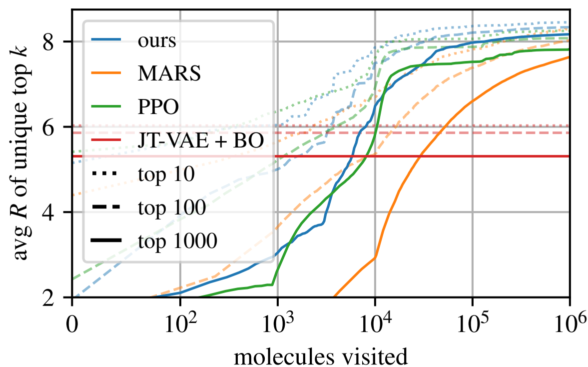


Figure 5.9: The average reward of the top-k as a function of learning (averaged over 3 runs). Only unique hits are counted. Note the log scale. Our method finds more unique good molecules faster.

method	Reward at 10^5 samples top-10	top-100	top-1000
GFlowNet	8.36 ± 0.01	8.21 ± 0.03	7.98 ± 0.04
MARS	8.05 ± 0.12	7.71 ± 0.09	7.13 ± 0.19
PPO	8.06 ± 0.26	7.87 ± 0.29	7.52 ± 0.26
	Reward at 10^6 samples		
GFlowNet	8.45 ± 0.03	8.34 ± 0.02	8.17 ± 0.02
MARS	8.31 ± 0.03	8.03 ± 0.08	7.64 ± 0.16
PPO	8.25 ± 0.12	8.08 ± 0.12	7.82 ± 0.16
	Reward for 10^6-equivalent compute		
JT-VAE + BO	6.03	5.86	5.31

Table 5.1: Average reward of the top-k molecules. Means and standard deviations computed over 3 runs. We see that GFlowNet produces significantly better molecules.

5.3.2.3 Experimental results

In Fig. 5.8 we show the empirical distribution of rewards in two settings; first when we train our model with $R(x)$, then with $R(x)^\beta$. If GFlowNet learns a reasonable policy π, this should shift the distribution to the right. This is indeed what we observe. We also compare GFlowNet to MARS (Xie et al., 2021), which is known to work well in the molecule domain, and observe the same shift. Note that GFlowNet finds more high reward molecules than MARS with these β values; this is consistent with the hypothesis that it finds more high-reward modes.

In Fig. 5.9, we show the average reward of the top-k molecules found so far, without allowing for duplicates (based on SMILES). We compare GFlowNet with MARS, PPO and JT-VAE (Jin et al., 2020) with Bayesian Optimization. As expected, PPO plateaus after a while; RL tends to be satisfied with good enough trajectories unless it is strongly regularized with exploration mechanisms. For GFlowNet and for MARS, the more molecules are visited, the better they become, with a slow convergence towards the proxy's max reward. Given the same compute time, JT-VAE+BO generates only about 10^3 molecules (due to its expensive Gaussian Process) and so does not perform well. We also report these numerical results in Table 5.1.

The maximum reward in the proxy's dataset is 10, with only 233 examples above 8. In our best run, we find 2339 unique molecules during training with a score above 8, 39 of which are in the dataset. We compute the average pairwise Tanimoto similarity (a standard measure of chemical similarity, see Bajusz et al., 2015) for the top 1000 samples: GFlowNet has a mean of 0.44 ± 0.01, PPO, 0.62 ± 0.03, and MARS, 0.59 ± 0.02 (mean and std over 3 runs). A random agent for this environment would yield an average pairwise

similarity of 0.231 (and very poor rewards). As expected, our MCMC baseline (MARS) and RL baseline (PPO) find less diverse candidates.

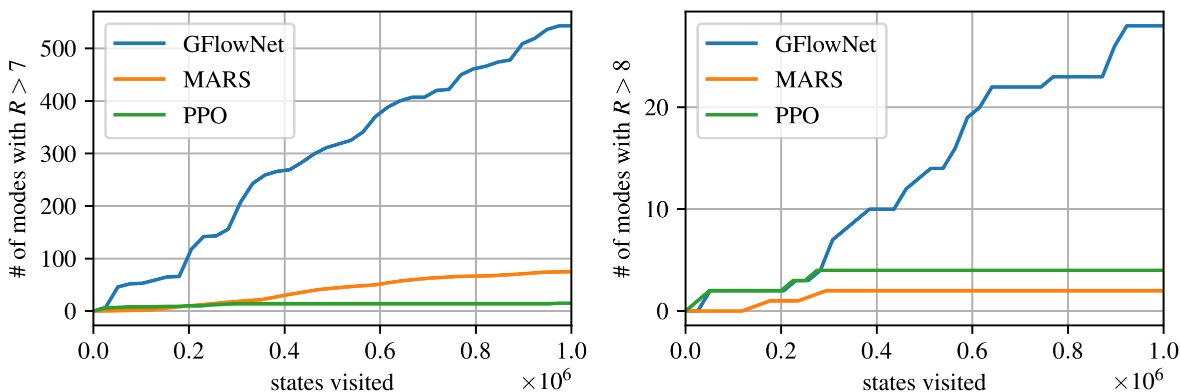


Figure 5.10: Number of Tanimoto-separated modes found above reward threshold T as a function of the number of molecules seen. See main text. Left, $T = 7$. Right, $T = 8$.

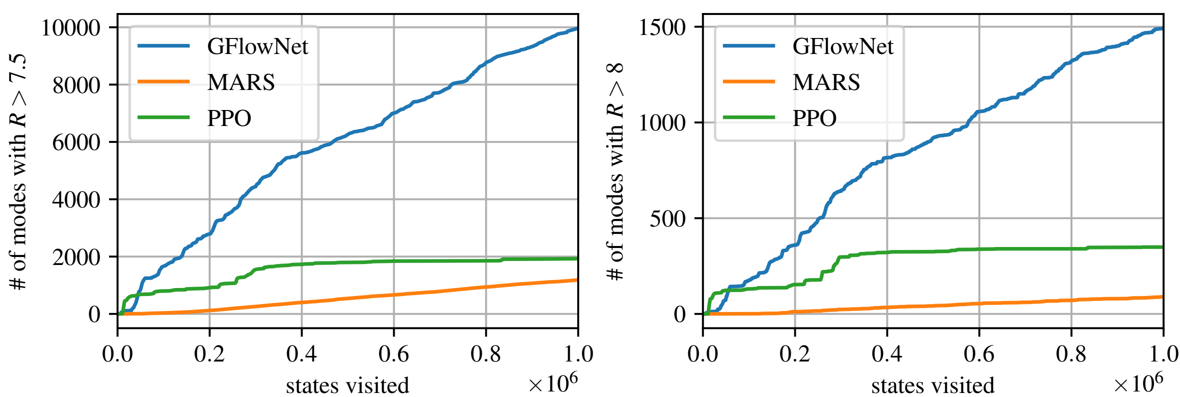


Figure 5.11: Number of diverse Bemis-Murcko scaffolds found above reward threshold T as a function of the number of molecules seen. Left, $T = 7.5$. Right, $T = 8$.

We can also see that GFlowNet produces much more diverse molecules by approximately counting the number of Tanimoto-separated modes found within the high-reward molecules. Here, we define "modes" as molecules with an energy above some threshold T, at most similar to each other in Tanimoto space at threshold S. In other words, we consider having found a new mode representative when a new molecule has a Tanimoto similarity smaller than S to every previously found mode's representative molecule. We choose a Tanimoto similarity S of 0.7 as a threshold, as it is commonly used in medicinal chemistry to find similar molecules, and a reward threshold of 7 or 8. We plot the results in Figure 5.10. We see that for $R > 7$, GFlowNet discovers many more modes than MARS or PPO, over 500, whereas MARS only discovers less than 100.

Another way to approximate the number of modes is to count the number of diverse Bemis-Murcko scaffolds (Bemis and Murcko, 1996) present within molecules above a certain reward threshold. We plot these counts in Figure 5.11, where we again see that GFlowNet finds a greater number of modes.

Next, let's try to understand what is learned by GFlowNet. In a large scale domain without access to $p(x)$, it is non-trivial to demonstrate that $u_\theta(x)$ matches the desired distribution $p(x) \propto R(x)$. This is due to the many-paths problem: to compute the true $p_\theta(x)$ we would need to sum the $p_\theta(\tau)$ of all the trajectories that lead to x, of which there can be an extremely large number. Instead, we show various measures that suggest that the learned distribution is consistent with the hypothesis the $u_\theta(x)$ matches $p(x) \propto R(x)^\beta$ well enough.

In Figure 5.12 we show how F_θ partially learns to match $R(x)$. In particular we plot the inflow of leaves (i.e. for leaves s' the $\sum_{s,a:T(s,a)=s'} F(s,a)$) as versus the target score $(R(x)^\beta)$.

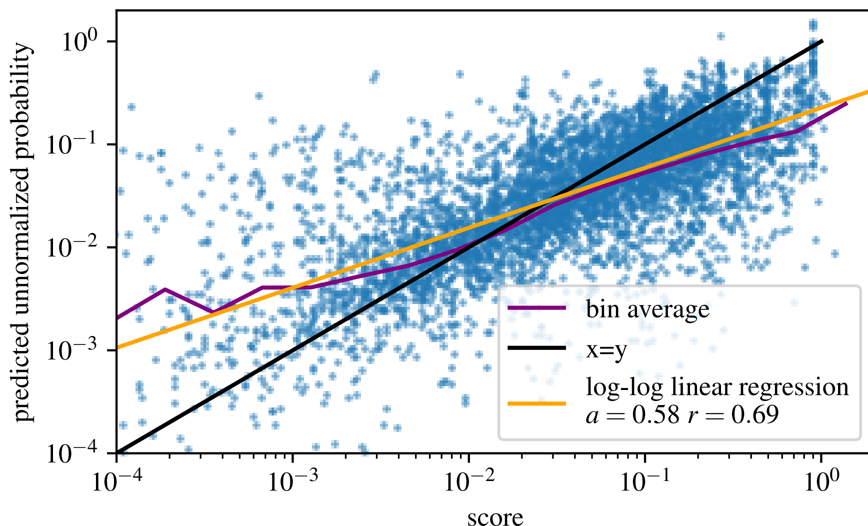


Figure 5.12: Scatter of the score $(R(x)^\beta)$ vs the inflow of leaves (the predicted unnormalized probability). The two should match. We see that a log-log linear regression has a slope of 0.58 and a r of 0.69. The slope being less than 1 suggests that GFlowNet tends to underestimate high rewards (this is plausible since high rewards are visited less often due to their rarity), but nonetheless reasonably fits its data. Here $\beta = 10$. We plot here the last 5k molecules generated by a run.

Another way to view that the learned probabilities are self-consistent is that the histograms of $R(x)/Z$ and $\hat{p}_\theta(x)/Z$ match, where we use the predicted $Z = \sum_{a\in\mathcal{A}(s_0)} F(s_0, a)$, and $\hat{p}_\theta(x)$ is the inflow of the leaf x as above. We show this in Figure 5.13.

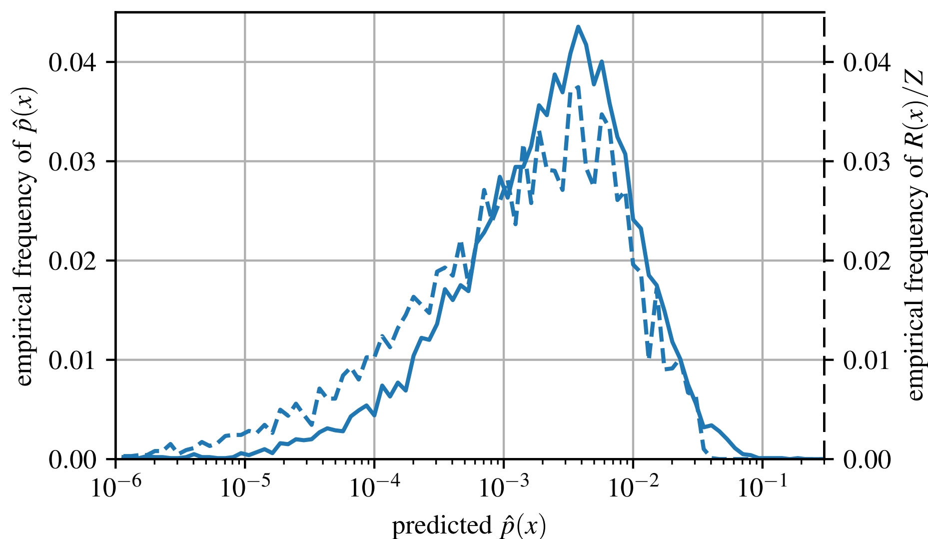


Figure 5.13: Histogram of the predicted density vs histogram of reward. The two should match. We compute these with the last 10k molecules generated by a run. This plot again suggests that the model is underfitted. It thinks the low-reward molecules are less likely than they actually are, or vice-versa that the low-reward molecules are better than they actually are. This is consistent with the previous plot showing a lower-than-1 slope.

In terms of loss, it is interesting that our models behaves similarly to value prediction in deep RL, in the sense that the value loss never goes to 0. This is somewhat expected due to bootstrapping, and the size of the state space. Indeed, in our hyper-grid experiments the loss does go to 0 as the model converges. We plot the loss separately for leaf transitions (where the inflow is trained to match the reward) and inner flow transitions (at visited states, where the inflow is trained to match the outflow) in Figure 5.14.

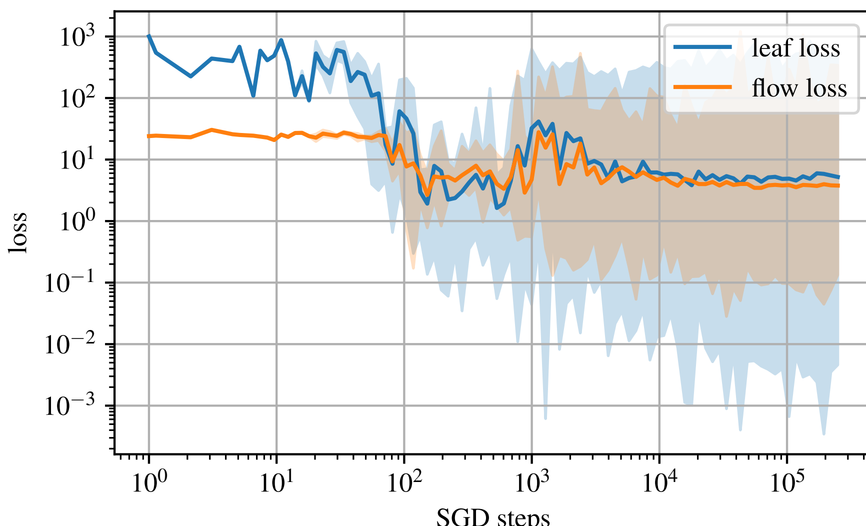


Figure 5.14: Loss as a function of training for a typical run of GFlowNet on the molecule domain. The shaded regions represent the min-max over the interval. We note several phases: In the initial phase the scale of the predictions are off and the leaf loss is very high. As prediction scales adjust we observe the second phase where the flow becomes consistent and we observe a dip in the loss. Then, as the model starts discovering more interesting samples, the loss goes up, and then down as it starts to correctly fit the flow over a large variety of samples. The lack of convergence is expected due to the massive state space; this is akin to value-based methods in deep RL on domains such as Atari.

5.3.3 Multi-Round Active Learning Experiments

To demonstrate the importance of diverse candidate generation in an active learning setting, we consider a sequential acquisition task. We simulate the setting where there is a limited budget for calls to the true oracle O. We use a proxy M initialized by training on a limited dataset of $(x, R(x))$ pairs D_0, where $R(x)$ is the true reward from the oracle. The generative model (u_θ) is trained to fit to the unnormalized probability function learned by the proxy M. We then sample a batch $B = \{x_1, x_2, \dots x_k\}$ where $x_i \sim u_\theta$, which is evaluated with the oracle O. The proxy M is updated with this newly acquired and labeled batch, and the process is repeated for N iterations.

More concretely, Algorithm 1 defines the procedure to train F_θ used in the inner loop of the multi-round experiments in the hyper-grid and molecule domains. The effect of diverse generation becomes apparent in the multi-round setting. Since the proxy itself is trained based on the input samples proposed by the generative models (and scored by the oracle, e.g., using docking), if the generative model is not exploratory enough, the reward (defined by the proxy) would only give useful learning signals around the discovered modes. The oracle outcomes $O(x)$ are scaled to be positive, and a hyper-parameter β (a kind of inverse

temperature) can be used to make the modes of the reward function more or less peaked.

Algorithm 1 Multi-Round Active Learning

Input:
Initial dataset $D_0 = \{x_i, y_i\}, |D_0| = n$;
k for top-k evaluation;
number of rounds (outer loop iterations) N;
Oracle O;
inverse temperature β
Returns: A set top-k(D_N) of high valued x
Initialize:
Proxy M;
Flow model Q_θ;
$i = 1$
while $i <= N$ **do**
Fit M on dataset D_{i-1}
Train F_θ with unnormalized probability function $r(x) = M(x)^\beta$ as target reward
Sample query batch $B = \{x_1, \ldots, x_b\}$ with $x_i \sim u_\theta$
Evaluate batch B with O, $\hat{D}_i = \{(x_1, O(x_1)), \ldots, (x_b, O(x_b))\}$
Update dataset $D_i = \hat{D}_i \cup D_{i-1}$
$i = i + 1$
end while

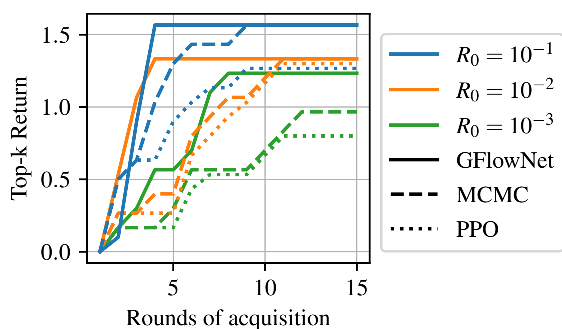


Figure 5.15: The top-k return (mean over 3 runs) in the 4-D Hyper-grid task with active learning. GFlowNet gets the highest return faster.

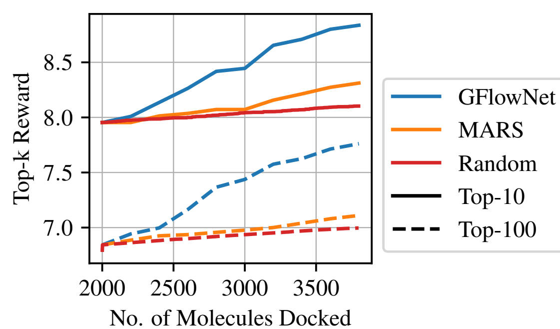


Figure 5.16: The top-k docking reward (mean over 3 runs) in the molecule task with active learning. GFlowNet consistently generates better samples.

Hyper-grid domain We present results for the multi-round task in the 4-D hyper-grid domain in Figure 5.15. We use a Gaussian Process (Williams and Rasmussen, 1995) as the proxy. We compare the *top-k Return* for all the methods, which is defined as $\text{mean}(\text{top-}k(D_i)) - \text{mean}(\text{top-}k(D_{i-1}))$, where D_i is the dataset of points acquired until step i, and $k = 10$ for this experiment. The initial dataset D_0 ($|D_0| = 512$) is the same for all the methods compared.

We use the Gaussian Process implementation from `botorch`[3] for the proxy. The query batch size of samples generated after each round is 16. The hyper-parameters for training the generative models are set to the best performing values from the single-round experiments.

The initial dataset only contains 4 of the modes. GFlowNet discovers 10 of the modes within 5 rounds, while MCMC discovered 10 within 10 rounds, whereas PPO manages to discover only 8 modes by the end (with $R_0 = 10^{-1}$).

We observe that GFlowNet consistently outperforms the baselines in terms of return over the initial set. We also observe that the mean pairwise L2-distance between the top-k points at the end of the final round is 0.83 ± 0.03, 0.61 ± 0.01 and 0.51 ± 0.02 for GFlowNet, MCMC and PPO respectively. This demonstrates the ability of GFlowNet to capture the modes, even in the absence of the true oracle, as well as the importance of capturing this diversity in multi-round settings.

Small Molecules For the molecule discovery task, we initialize an MPNN proxy to predict docking scores from AutoDock (Trott and Olson, 2010), with $|D_0| = 2000$ molecules. At the end of each round we generate 200 molecules which are evaluated with AutoDock and used to update the proxy. Figure 5.16 shows GFlowNet discovers molecules with significantly higher energies than the initial set D_0. It also consistently outperforms MARS as well as Random Acquisition. PPO training was unstable and diverged consistently so the numbers are not reported. The mean pairwise Tanimoto similarity in the initial set is 0.60. At the end of the final round, it is 0.54 ± 0.04 for GFlowNet and 0.64 ± 0.03 for MARS. This further demonstrates the ability of GFlowNet to generate diverse candidates, which ultimately helps improve the final performance on the task.

The initial set D_0 of 2000 molecules is sampled randomly from the 300k dataset. At each round, for the MPNN proxy retraining, we use a fixed validation set for determining the stopping criterion. This validation set of 3000 examples is also sampled randomly from the 300k dataset. We use fewer iterations when fitting the generative model, and the rest of the hyper-parameters are the same as in the single round setting.

	Reward after 1800 docking evaluations	
method	top-10	top-100
GFlowNet	8.83 ± 0.15	7.76 ± 0.11
MARS	8.27 ± 0.20	7.08 ± 0.13

[3] http://botorch.org/

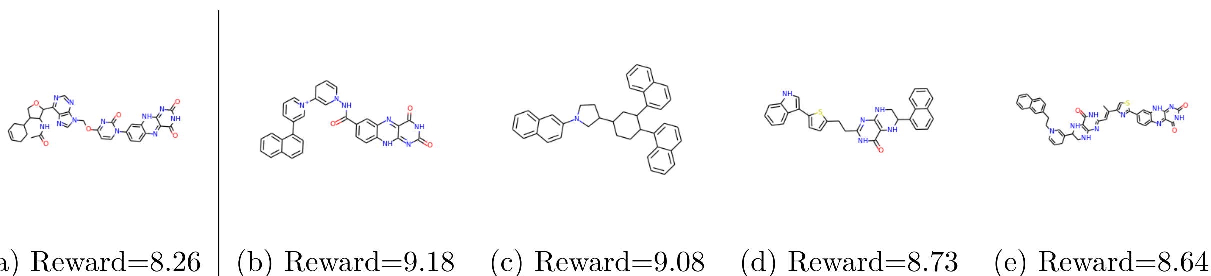


Figure 5.17: (a) Highest reward molecule in D_0 in the multi-round molecule experiments. (b) Highest Reward molecule generated by GFlowNet. (c)-(e) Samples from the top-10 molecules generated by GFlowNet.

5.3.4 Revisiting the hyper-grid with RL tricks

The objective of GFlowNet uses bootstrapping, it uses prediction of future events (the flow of outgoing edges in the DAG MDP) to learn present quantities (the flow of incoming edges). This has enough similarities with TD methods that it warrants trying to apply the methods that improve the performance of TD to GFlowNet.

In particular here, we try the following: using a frozen target (Mnih et al., 2013), an exponential moving average target (Lillicrap et al., 2016), or a doubly parameterized target (van Hasselt et al., 2016); using a replay buffer (Mnih et al., 2013), a prioritized replay buffer (Schaul et al., 2015), or a top-k replay buffer; using different optimizers, Adam (Kingma and Ba, 2015), RMSProp (Hinton et al., 2012), or momentum (Polyak, 1964); and using an ϵ-online sampling strategy (Mnih et al., 2013), or a higher softmax sampling temperature for exploration.

We revisit the hyper-grid setting of § 5.3.1 with $n = 4$ and $H = 8$. In the figures that follow, each setting has 20 seeds, and we plot the histogram of mean average distributional error over the seeds, $\mathbb{E}[|u_\theta(x) - p(x)|]$; both $u_\theta(x)$ and p are computed exactly since the environment is small.

First we compare optimizers in Figure 5.18, and somewhat unsurprisingly find that Adam and RMSProp are close contenders, with Adam having less variance (Henderson et al., 2018). Momentum SGD lags far behind, as it lacks the advantage of per-parameter adaptive learning rates.

We compare different replay mechanisms in Figure 5.19. Interestingly we find that both uniform sampling (Mnih et al., 2013) and prioritized experience replay (Schaul et al., 2015) help quite a bit. This suggests that the optimal data distribution to learn flows is not necessarily $u_\theta(x)$, but also possibly that the i.i.d.-ness induced by the replay mechanism (i.i.d. *transitions* are sampled) is superior to sampling i.i.d. trajectories from u_θ. For the no-replay setting, a minibatch is formed with 16 trajectories sampled from u_θ, as such

there is still some correlation within the batch.

We also compare to a replay buffer that replays the top-k trajectories seen so far. Figure 5.20 compares several settings of this replay style, whereby n_{mb} trajectories are used in the minibatch of size 16 (trajectories), sampled from the best k trajectories seen so far.

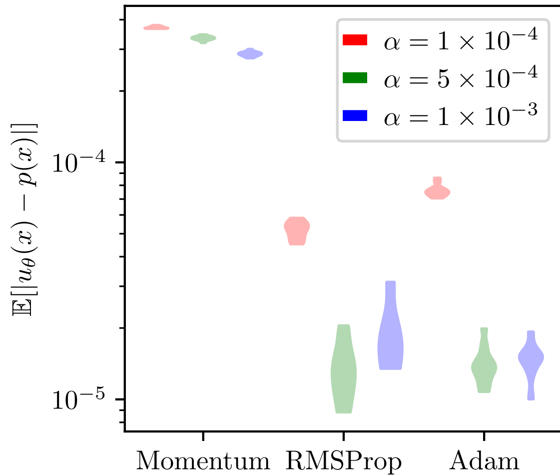


Figure 5.18: Hyper-grid, $n = 4$, $H = 8$, Momentum SGD, RMSProp, and Adam compared with learning rates α. y-axis: the estimated distribution's L1 error.

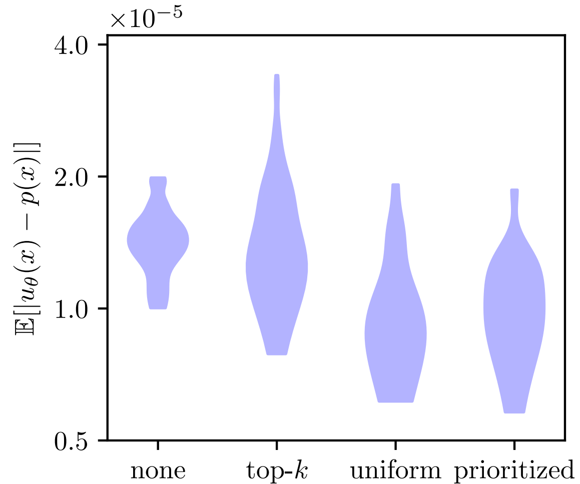


Figure 5.19: Hyper-grid, $n = 4$, $H = 8$, no replay, top-k replay, uniform, and prioritized experience replays compared. y-axis: the estimated distribution's L1 error.

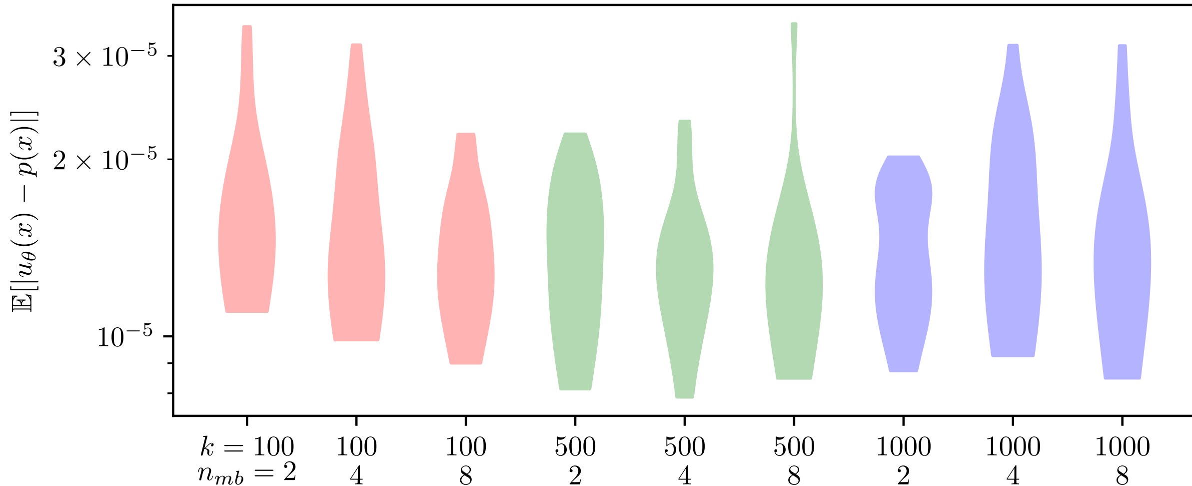


Figure 5.20: Hyper-grid, $n = 4$, $H = 8$, top-k replay buffer of size k mixing n_{mb} top trajectories in the minibatch (of size 16). We only use color as a visual aid.

We then compare different uses of target networks, but find that, unlike reported in deep RL, no method beats *not* using any separate target network. This may however be due to how easy this setting is. Figure 5.21 compares no-target, using an exponential

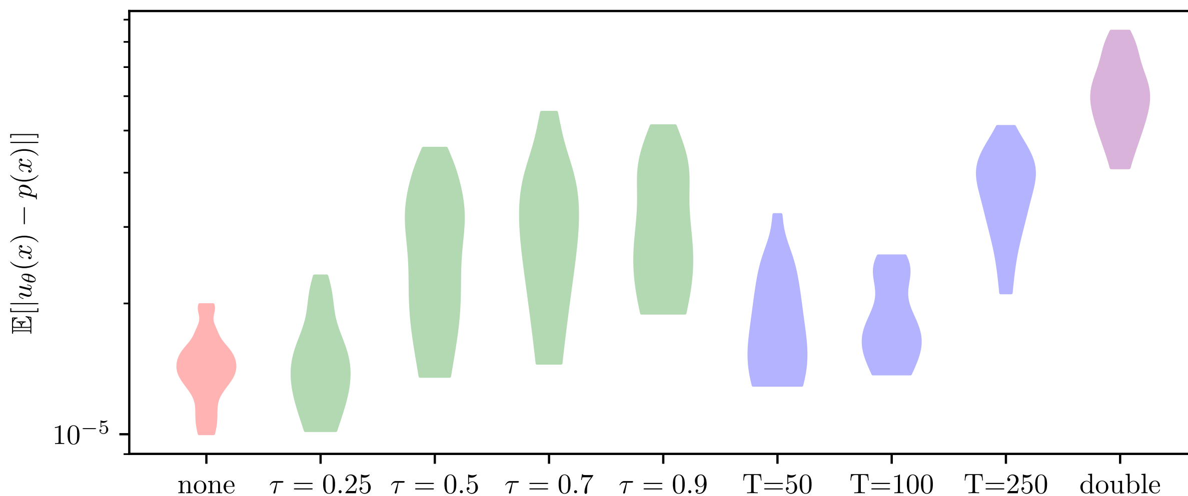


Figure 5.21: Hyper-grid, $n = 4$, $H = 8$, comparing no target, a moving average target with rate τ, a frozen target with update period T, and a doubly parameterized target. We only use color as a visual aid.

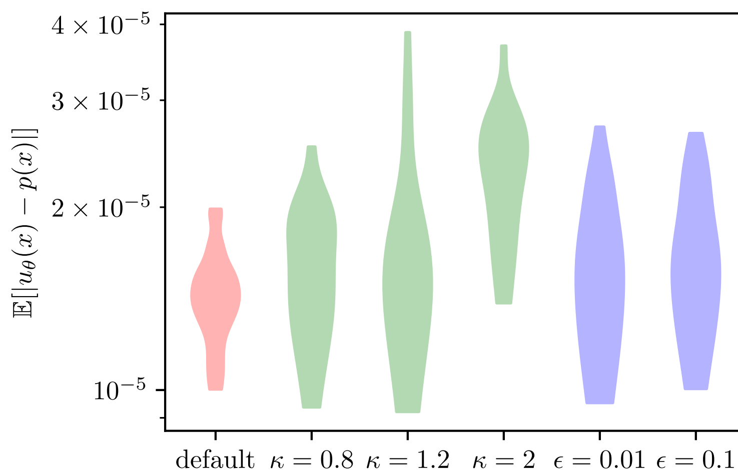


Figure 5.22: Hyper-grid, $n = 4$, $H = 8$, comparing no exploration, using a softmax temperature of κ, and sampling a random action with probability ϵ. We only use color as a visual aid.

moving average target (Lillicrap et al., 2016) with rate τ ($\theta'_t = \tau\theta'_{t-1} + (1-\tau)\theta_t$), a frozen target (Mnih et al., 2013) updated after every T SGD steps, and a doubly parameterized target (van Hasselt et al., 2016).

Finally, we compare different ways of exploring the state space. When computing the policy since we predict $F^{\log}$, we obtain π by taking the softmax of $F^{\log}$. By multiplying the logits by κ we can push the policy to be more or less exploratory. As in Mnih et al. (2013), we also try exploring via sampling a random action at every step with probability ϵ. We see that these approaches can have beneficial effects, but can also hurt, as seen in the higher variance of the error distributions.

5.4 Discussion & Limitations

In this chapter we have introduced a novel TD-like objective for learning a flow for each state and state-action pair such that policies sampling actions proportional to these flows draw terminal states in proportion to their reward. This can be seen as an alternative approach to turn an energy function into a fast generative model, without the need for an iterative method like that needed with MCMC methods, and with the advantage that when training succeeds, the policy generates a great diversity of samples near the main modes of the target distribution without being slowed by issues of mixing between modes.

One downside of the proposed method is that, as for TD-based methods, the use of bootstrapping may, as we have seen in previous chapters, cause optimization challenges (Kumar et al., 2020) and limit its performance.

In applications like drug discovery, sampling from the regions surrounding each mode is already an important advantage, but future work should investigate how to combine such a generative approach to local optimization in order to refine the generated samples and approach the local maxima of reward while keeping the batches of candidates diverse. As it is, our approach is not necessarily suitable for needle-in-a-haystack scenarios, as while it is capable of identifying many modes, it seems to rely on these modes being at least partially locally concave. In other words, it relies on local maxima to have close neighbours with similarly high rewards, and is likely to samples those neighbours rather than the actual maxima, especially if there are many of those neighbours.

Finally, we believe that the proposed framework has the potential to be used in a number of scenarios. In general, any black-box optimization problem where candidates are generated iteratively may benefit from this approach: material discovery, antibiotic discovery, hyperparameter search, or reasoning tasks all could benefit from reward-proportional sampling.

6

Learning to Act and Understand Without Supervision

This chapter synthesizes my contributions to a novel way to learn representations in RL in the absence of reward (Bengio et al., 2017; Thomas et al., 2017, 2018).

In the previous chapters, we've discussed methods that train Deep Neural Networks to accomplish certain things, evaluate the goodness of a policy, learn to solve problems, learn to generate interesting objects. All of these depend on a *reward signal*, which, loosely, quantifies the goodness of a state. In this chapter, we attempt to provide methods that still learn interesting things, behaviours and representations, in the **absence of reward**. In most environments used in RL, this reward is something that is crafted, designed to achieve some desired behaviour from agents; in other words, reward is a form of supervision. From this point of view, what we present next may well be qualified of *unsupervised reinforcement learning*.

More specifically, we present a method that learns a set of policies to which corresponds a set of representations (a designated layer within a DNN). These policies are driven to learn the different *controllable* aspects of the environment in which an agent is situated. Moreover, these policies are driven to learn not to interfere with each other, and learn *independent* factors of variation, which are in turn represented through a learning neural network. This ensemble is dubbed **Independently Controllable Features**.

In the next sections, we loosely follow the original material on controllable features (Bengio et al., 2017; Thomas et al., 2017, 2018), with some credit again going to my coauthors for their contributions.

6.1 Unsupervised Learning

Unsupervised learning refers to a broad class of algorithms which learn in the absence of supervision. Typically, this terminology is used to contrast with supervised learning, where one trains models of some mapping $y = f(x)$ with labelled data pairs (x, y).

With only access to observations, a set of x, one can nonetheless do interesting things. For example, one can cluster observations based on their similarity (Forgy, 1965; Lloyd,

1982), learn a likelihood model of the data (Ferguson, 1973; Rasmussen et al., 1999), or extract the principal factors of variation (Pearson, 1901; Jutten and Herault, 1991). These can help us understand better, or provide a valuable signal that can be used for downstream supervised learning.

6.1.1 Disentanglement and Representation Learning

In Deep Learning, something particularly interesting can be done with unlabeled data: pre-training models. Recall that DNNs are made of a series of layers, which when presented with an input compute a hidden feature, or representation for this input. This representation is typically taken to be the activations of the second to last layer, i.e. $h_{L-1}(\mathbf{x})$ following the notation of §2.1.2, although as we will see we can consider the entire network to be producing a representation and thus take it to be last layer's output. In such a case we say the model is an **encoder**. For what follows we will shorthand $h_L(\mathbf{x})$ with $\mathbf{h}$.

A common way to leverage access to large amounts of unlabeled data is to train $\mathbf{h}$ to have certain desirable properties. In fact, before we knew how to train very deep networks, training each layer to have desirable properties before finetuning (more on this later) the entire network was the best way to train DNNs with more than a few layers (Hinton et al., 2006; Bengio et al., 2007).

One common strategy to train $\mathbf{h}$ is to train an **autoencoder**. Autoencoders work by reducing their input to a vector with a smaller number of dimensions, thus forcing some sort of data compression to occur (Hinton and Salakhutdinov, 2006; Vincent et al., 2008). Formally, an autoencoder consists of an encoder $f_\theta : \mathbb{R}^{d_{in}} \to \mathbb{R}^{d_{out}}$ and a decoder $g_{\theta'} : \mathbb{R}^{d_{out}} \to \mathbb{R}^{d_{in}}$. Here we consider that $\mathbf{h} = f_\theta(\mathbf{x})$, i.e. in an autoencoder the representation is the last layer's. A plain autoencoder is trained to minimize the loss:

$$\mathcal{L}_{AE} = \mathbb{E}_{\mathbf{x}} \|\mathbf{x} - g_{\theta'}(f_\theta(\mathbf{x}))\|, \tag{6.1}$$

by adapting both θ and θ'. When the dimension of $\mathbf{h}$ is smaller than that of $\mathbf{x}$, this can be seen as a compression mechanism, but other mechanisms can induce compression. For example denoising autoencoders (Vincent et al., 2008) can be seen as doing a contractive compression of the representation space (Rifai et al., 2011) by minimizing the norm of $g(f(\mathbf{x} + \epsilon)) - \mathbf{x}$, where ϵ is some appropriate noise.

Often, this compressive force pushes the optimization procedure to uncover principal factors of variation of the data on which they are trained. However, this does not necessarily imply that the different components of the vector $\mathbf{h} = f(\mathbf{x})$ are *individually meaningful*. In fact, note that for any bijective transformation T, we could obtain the same reconstruction error by replacing f by $T \circ f$ and g by $g \circ T^{-1}$, so we should not expect any form of

disentangling of the factors of variation unless some additional constraints or penalties are imposed on h.

Generally speaking, disentanglement occurs when the different quantities that *explain* how data are generated are made explicit. For example, in a dataset of images of red or blue squares and circles, one could disentangle the pixels into 4 factors of variation, the color, the shape, and the horizontal and vertical position of the object on the image.

There are several other ways to discover and disentangle underlying factors of variation. Many deep generative models, including variational autoencoders (Kingma and Welling, 2014) and other descendants of the Helmholtz machine (Dayan et al., 1995), generative adversarial networks (Goodfellow et al., 2014) or non-linear versions of ICA (Dinh et al., 2014; Hyvarinen and Morioka, 2016) attempt to disentangle the underlying factors of variation by assuming that their joint distribution (marginalizing out the observed x) factorizes, i.e., that they are marginally independent.

Once some representation has been learned on unlabeled data, i.e. some set of parameters θ has been learned for the encoder f, it is possible to *finetune* (Hinton et al., 2006) θ to perform some supervised task by training it on labeled data. Finally, it is possible to both use an unsupervised representational objective and a supervised objective are used at the same time (on possibly disjoint datasets), this is described as *semi-supervised* learning.

6.1.2 Reinforcement Learning without Reinforcement

In the previous section we've seen that it is possible to learn without supervision when given data points with structure. In RL, not only do we have access to individual data points, the observations of each state, we also have access to trajectories. That is, we have access to the *temporal structure* hidden in these data through trajectories. Furthermore, since agents have agency, they can in theory choose which regions of the state space to visit and be "curious" about the world (Schmidhuber, 1991; Singh et al., 2005; Still and Precup, 2012).

Leveraging this temporal structure yields many opportunities to indirectly learn good internal representations without requiring explicit supervision. In the work Jaderberg et al. (2017), agents learn off-policy to control their pixel inputs, pushing them to learn features that, intuitively, help control the environment (at the pixel level). Oh et al. (2015) propose models that learn to predict future observations, conditioned on action sequences, which pushes the agent's internal representations to capture temporal features. Many more works go in this direction, such as (deep) successor feature representations (Dayan, 1993; Kulkarni et al., 2016) which learn to predict how internal representations activate and accumulate through time, or the options framework (Sutton et al., 1999b; Precup,

2000) when used in conjunction with neural networks (Bacon et al., 2016) which aims to decompose the behaviour of an agent into a discrete set of general and reusable policies, the so-called options.

The approach we propose next is also similar in spirit to the Horde architecture (Sutton et al., 2009a). In that scenario, agents learn policies that maximize specific inputs, whereas we learn policies that control simultaneously learned features of the input. The predictions for all these policies then become features for the agent.

6.2 Jointly Learning Features and Policies

In the previous section we've described unsupervised learning, and suggested that there are many approaches to learn representations and policies in the absence of a reward signal in RL. Interestingly, the previous approaches that we highlighted mostly focus on one or the other, either learning a representation or learning sets of policies. Here we propose instead a method that explicitly learns a joint set of features and policies, by aiming to make them **independent**.

To do this, we assume that there are factors of variation underlying the observations coming from an interactive environment that are *independently controllable*. That is, a controllable factor of variation is one for which there exists a policy which will modify that factor only, and not the others. For example, the object associated with a set of pixels could be acted on independently from other objects, which would explain variations in its pose and scale when we move it around while leaving the others generally unchanged. The object position in this case is a *factor of variation*. What poses a challenge for discovering and mapping such factors into computed features is the fact that the factors are not explicitly observed. Our goal is for the agent to autonomously discover such factors – which we call **independently controllable features** – along with policies that control them. While these may seem like strong assumptions about the nature of the environment, we argue that these assumptions are similar to regularizers, and are meant to make a difficult learning problem (that of learning good representations which disentangle underlying factors) better constrained.

6.2.1 Quantifying Independence into Learning Objectives

What does it mean for features to be independent in a control setting? There are many ways in which this desire can be interpreted. We propose a particular formulation which binds features and policies together to express independence.

Consider an autoencoder (f, g), which produces K features, i.e. $f(\mathbf{x}) \in \mathbb{R}^K$, $f_k(\mathbf{x}) \in \mathbb{R}$. In tandem with these features we train K policies, denoted $\pi_k(a|s)$, that map an agent's observation s to a categorical distribution over a set of actions a. Autoencoders can learn relatively arbitrary feature representations, but we would like many of these features to correspond to controllable factors in the learner's environment. Specifically, we would like policy π_k to cause a change only in f_k and not in any other features (we relax this constraint later on in §6.2.2). We think of f_k and π_k as a feature-policy pair.

In order to quantify the change in f_k when actions are taken according to π_k, we define the *selectivity* of a feature as:

$$sel(s, a, k) = \mathbb{E}_{s' \sim \mathcal{P}^a_{ss'}} \left[\frac{|f_k(s') - f_k(s)|}{\sum_{k'} |f_{k'}(s') - f_{k'}(s)|} \right]. \tag{6.2}$$

where s,s' are successive raw state representations (e.g. pixels), a is the action, and $\mathcal{P}^a_{ss'}$ is the environment transition distribution from s to s' under action a. The normalization factor in the denominator of the above equation ensures that the selectivity of f_k is maximal when *only that single feature* f_k changes as a result of some action.

By having an objective that maximizes selectivity *and* minimizes the autoencoder objective, we can ensure that the features learned can both capture the main factors of variation in the data and recover independently controllable factors. Hence, we define the following objective, which can be minimized jointly on π_k, f and g, via stochastic gradient descent:

$$\underbrace{\mathbb{E}_s[\tfrac{1}{2}||s - g(f(s))||_2^2]}_{\mathcal{L}_{ae} \text{ the reconstruction error}} \; - \underbrace{\lambda \sum_k \mathbb{E}_s[\sum_a \pi_k(a|s) sel(s, a, k)]}_{\mathcal{L}_{sel} \text{ the disentanglement objective}}. \tag{6.3}$$

Here one can think of $sel(s, a, k)$ as the reward signal $R_k(s, a)$ of a control problem, and the expected reward $\mathbb{E}_{a \sim \pi_k}[R_k]$ is maximized by finding the optimal set of policies π_k.

Note that many variations of this objective are possible. For example it is also possible to have *directed* selectivity: by using $\max\{0, f'_k - f_k\}$ (denoted $|f'_k - f_k|_+$) or simply $f'_k - f_k$ instead of the absolute value $|f'_k - f_k|$ in the numerator of (6.2), the policies must learn to increase the learned latent feature rather than simply change it. This may be useful if the policy to gradually increase a feature is distinct from the policy that decreases it. Using log-selectivity, $\log sel$, or this sharpened form, $\log(sel/(1 - sel))$, may also lead to easier optimization.

Another variation is to consider the return instead of the immediate reward. Indeed, we may want to incentivize agents that modify aspects of their environments over multiple steps rather than a single one.

The learning algorithm we propose is summarized in Algorithm 2, where θ_f and θ_g are the parameters of f, g and θ_k the parameters of π_k.

Algorithm 2 Training an autoencoder with disentangled factors

1: **for** $t = 1..T$ **do**
2: Sample s from the environment
3: $\theta_f \leftarrow \theta_f - \eta_f \nabla_{\theta_f}[\frac{1}{2}||s - g(f(s))||_2^2]$
4: $\theta_g \leftarrow \theta_g - \eta_g \nabla_{\theta_g}[\frac{1}{2}||s - g(f(s))||_2^2]$
5: **for** $k = 1..n$ **do**
6: $\theta_f \leftarrow \theta_f + \eta_f\ \lambda\ \nabla_{\theta_f}\mathbb{E}_{a\sim\pi_k(\cdot|s)}[sel(s, a, k)]$
7: $\theta_k \leftarrow \theta_k + \eta_k\ \lambda\ \nabla_{\theta_k}\mathbb{E}_{a\sim\pi_k(\cdot|s)}[sel(s, a, k)]$
8: **end for**
9: **end for**

The gradients on lines 3 and 4 are computed exactly via backpropagation. In our experiments, the gradient on line 6 is also computed by backpropagation and sampling of the expectation, while the gradient on line 7 is computed with the REINFORCE (Glynn, 1987; Williams, 1992b) estimator:

$$\nabla_{\theta_k}\mathbb{E}_{a\sim\pi_k(\cdot|s)}[sel(s, a, k)] = \mathbb{E}_{a\sim\pi_k(\cdot|s)}[(sel(s, a, k) - b(s)) \cdot \nabla_{\theta_k} \log \pi_k(a|s)],$$

where $b(s)$ is a baseline function, which can for example be chosen to be the mean reward or an estimate of the value of the state.

6.2.2 Expanding controllable factors to be continuous

A limitation of the approach in Algorithm 2 is that it requires the set of potentially controllable factors to be small and enumerated. This makes sense in a simple environment where we always have the same set of objects in the scene. But in more realistic environments, the number of possible objects present in the set can be combinatorially large (and better described by notions such as types), while an individual scene will only comprise a finite number of *instances* of such objects. Therefore, instead of indexing the possible factors by an integer, we propose to index them by an embedding, i.e., a real-valued vector.

In what we defined above, we enforced variations in the environment to be captured by a coordinate of $h = f(s)$. We can view this as having a set of k attribute variations $A(h' - h, k) = |h' - h|_k$ who are influenced separately by the policies π_k. We now relax this assumption by indexing this set by a learned real-valued vector ϕ leading to a continuous set of attributes $A(h' - h, \phi) \in \mathbb{R}$. The idea of mapping symbolic entities to a distributed representation is one of the key ingredients of the success of deep learning (Goodfellow et al., 2016), and can be exploited here as well.

6.2.2.1 Selecting attributes

Conditioned on a scene representation h, a distribution of policies are feasible. Samples from this distribution represent ways to modify the scene and thus may trigger an internal selectivity reward signal. For instance, h might represent a room with objects such as a light switch. $\phi = \phi(h, z)$ can be thought of as the distributed representation for the "name" of an underlying factor, to which is associated a policy and a value. In this setting, the light in a room could be a factor that could be either on or off. It could be associated with a policy to turn it on, and a binary value referring to its state, called an attribute or a feature value.

We wish to jointly learn the policy $\pi_\phi(\cdot|s)$ that modifies the scene, so as to control the corresponding value of the attribute in the scene, whose variation is computed by an attribute variation selector function $A(h'-h, \phi) \in \mathbb{R}$. In order to get a distribution of such embeddings, we compute $\phi(h, z)$ as a function of h and some random noise z.

In this scenario, one strategy to determine whether some selected attribute variation $A(h'-h, \phi)$ evolves *independently* from other attributes variations is to compare its value (in expectation over the policy actions) to the values obtained with other ϕ' factors. We thus compute the following selectivity that acts as an intrinsic reward signal, generalizing (6.2):

$$sel(h, \phi) = \mathbb{E}_{a\sim\pi_\phi(\cdot|s), s'\sim\mathcal{P}^a_{ss'}} \left[\frac{A(h'-h, \phi)}{\mathbb{E}_{\phi'}|A(h'-h, \phi')|} \right], \tag{6.4}$$

where $h' = f(s')$. We approximate the expectation over ϕ' by sampling a fixed number of factor embeddings. This model is then trained by jointly minimizing the autoencoder reconstruction cost $\mathcal{L}_{ae}$ and the disentanglement objective $\mathcal{L}_{sel}$ as depicted in Figure 6.1.

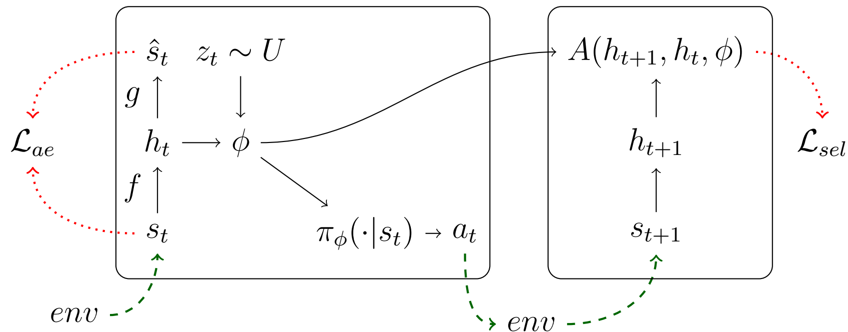


Figure 6.1: The proposed distributed representation architecture. $\mathcal{L}_{ae}$ and $\mathcal{L}_{sel}$ are the reconstruction and selectivity objectives respectively.

6.2.2.2 Implementing an attribute selector

Ideally $A(h' - h, \phi)$ could be an arbitrary function, e.g. a neural network, but such a function may be harder to optimize. Instead, we observe that in the discrete case mentioned previously, using $A(h' - h, \phi)$ to select attribute k is equivalent to $\phi^\top |h' - h|$ where ϕ is a one-hot vector at index k. One simple step towards continuous embeddings is to relax this constraint, and let ϕ be a function of h and random vector z, drawn from uniform distribution, and compute A as $A(h' - h, \phi) = \phi(h, z)^\top |h' - h|$. An even better form, used in most of our experiments, is to wrap this in a gaussian kernel: $A(h' - h, \phi) = \exp(-||h' - h - \phi||^2/(2\sigma^2))$ because of the better numerical stability it provides.

Unlike in the finite case, we are not sampling uniformly over policies π_k, as we now let a neural network choose ϕ's probability distribution. This could lead to exploration issues. We demonstrate that simple strategies allow for a network to learn simple distributions in the experiments of §6.3.3.

6.3 Empirical Results

In order to validate that our method learns independently controllable features, we perform several experiments. First, in the most basic gridworld-like setting, an agent is allowed to move around in four directions. This basic domain allows us to verify whether in the discrete case, the learning process disentangles the underlying features and recovers the ground truth properties of the environment.

Then, we show results of our continuous factors embeddings method applied to MazeBase (Sukhbaatar et al., 2015), as well as how we can use the learned representations to tackle policy inference or planning problems.

6.3.1 A simple gridworld

Our first experiment is performed on a gridworld-like setting, illustrated in Figure 6.2(a): the agent sees a 2×2 square on a 12×12 pixel grid, and has 4 actions that move it up, down, left or right. We use the following autoencoder architecture: f has two $16 \times 3 \times 3$ ReLU convolutional layers with stride 2, followed by a fully connected ReLU layer of 32 units, and a tanh layer of $n = 4$ features; g is the transpose architecture of f; π_k is a softmax policy over 4 actions, computed from the output of the ReLU fully connected layer. We use Adam (Kingma and Ba, 2014) to perform gradient descent.

By interacting with the environment, an autoencoder with directed selectivity (objective (6.2) without absolute value in the numerator) learns latent features that map to the

(x, y) position of the square (see Figure 6.2(b,c)), without ever having explicit access to these values, and while reconstructing its input properly. In contrast, a plain autoencoder also reconstructs properly but without learning the two latent (x, y) features explicitly.

Note that in this setting, the learning process is robust to a stochastic version of the environment – where with probability p either no action is taken ($s = s'$) or a random action is taken. We have successfully trained models recovering $\pm x$ and $\pm y$ with up to $p = 0.5$, using the same architecture but a smaller learning rate.

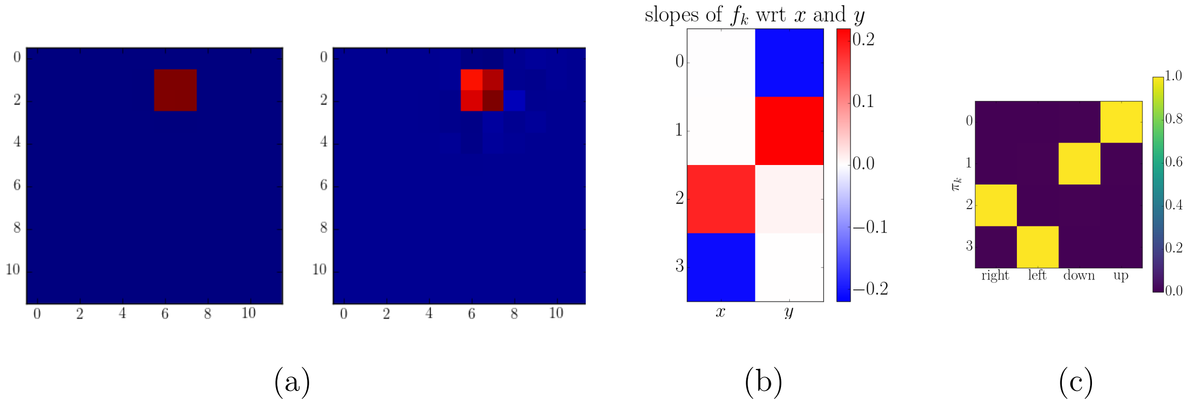


Figure 6.2: A simple gridworld with 4 actions that push a square left, right, up or down. (a) left is an example ground truth, right is the reconstruction of the model trained with selectivity. (b) The slope of a linear regression of the true features (the real x and y position of the agent) as a function of each latent feature. White is no correlation, blue and red indicate strong negative or positive slopes respectively. f_0 and f_1 recover y and f_2 and f_3 recover x. (c) Each row is a policy π_k, each column corresponds to an action (left/right/up/down). Cell (k, i) is the average over s of $\pi_k(a_i|s)$;

6.3.2 Selectivity as an only objective

We also find experimentally that training discrete independently controllable features without training the autoencoder objective correctly recovers ground truth features and their associated control policies. Albeit slower than when jointly training an autoencoder, this shows that the objective we propose is strong enough to provide a learning signal for discovering a disentangled latent representation.

We train such a model on a gridworld MNIST environment, where instead of a 2×2 square there are two MNIST digits . The two digits can be moved on the grid via 4 directional actions (so there are 8 actions total), the first digit is always odd and the second digit always even, so they are distiguishable. In Figure 6.3 we plot each latent feature f_k as a curve, as a function of each ground truth. For example we see that the

black feature recovers $+x_1$, the horizontal position of the first digit, or that the purple feature recovers $-y_2$, the vertical position of the second digit.

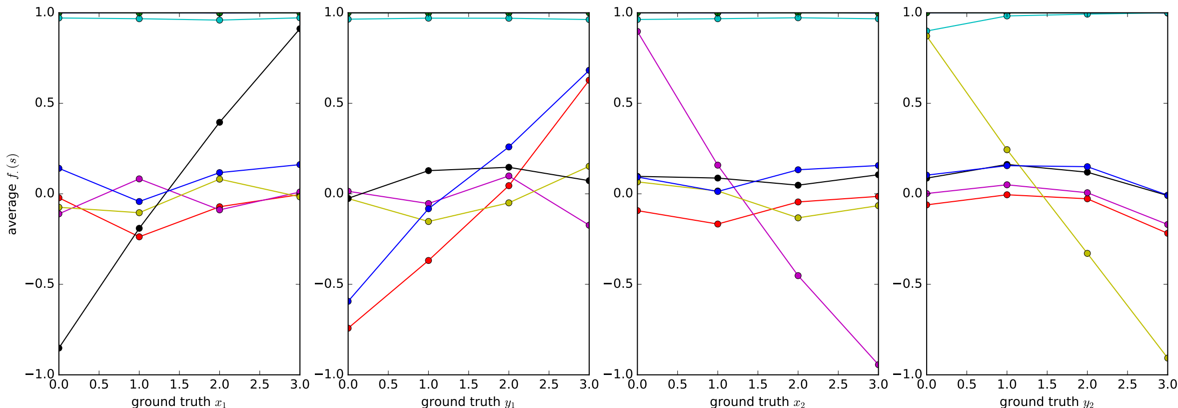


Figure 6.3: In a gridworld environment with 2 objects (in this case 2 MNIST digits), we know there are 4 underlying features, the (x_i, y_i) position of each digit i. Here each of the four plots represents the evolution of the f_k's as a function of their underlying feature, from left to right x_1, y_1, x_2, y_2. We see that for each of them, at least one f_k recovers it almost linearly, from the raw pixels only.

6.3.3 Experiments on MazeBase

We use MazeBase (Sukhbaatar et al., 2015) to assess the performance of our continuous embeddings approach on a more complex and well-known environment. MazeBase contains 10 different 2D games in which an agent has to solve a specific task (going to a certain location on the board, activate switches, move a block to a specific place, and so on). We do not aim to solve the game, and only deal with one-step policies.

In this setting, the agent (a red circle) can move in a small environment (64×64 pixels) and perform the actions `down, left, right, up`, and, to complexify the disentanglement task, we add the redundant action `up` as well as the action `down+left`. The agent can go anywhere except on the orange blocks.

In Figure 6.4, we show that the learned representation is such that for each underlying factor of variation, the learned representation clusters dh vectors such that it is possible to decompose the variation between two arbitrary state representations as a sum of small variations along a trajectory (Figure 6.5).

6.3.3.1 Continuous policy embeddings

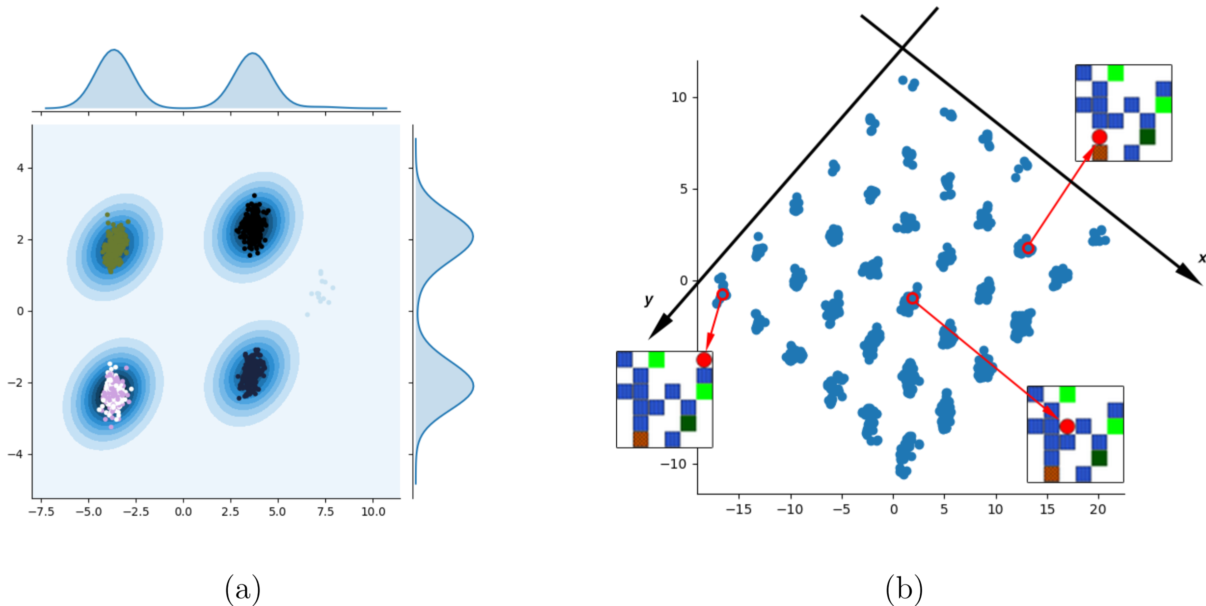
We consider the model described in §6.2.2.1. Our architecure is as follows: the encoder, mapping the raw pixel state to a latent representation, is a 4-layer convolutional neural

network with batch normalization (Ioffe and Szegedy, 2015) and leaky ReLU activations. The decoder uses the transposed architecture with ReLU activations. The noise z is sampled from a 6-dimensional gaussian distribution and both the generator $\Phi(h, z)$ and the policy $\pi(h, \phi)$ are neural networks consisting of 2 fully-connected layers. Our attribute selector $A(dh, \phi)$ is a gaussian kernel. In practice, a minibatch of $n = 64$ vectors $\phi_1, \ldots, \phi_{64}$ is sampled at each step. The agent randomly choses one $\phi = \phi_{behavior}$ and samples an action $a \sim \pi(h, \phi_{behavior})$. Our model parameters are then updated using policy gradient and importance sampling. For each selectivity reward, the term $\mathbb{E}_{\phi'}[|A(h' - h, \phi')|]$ is estimated as $\frac{1}{n}\sum_{i=1}^{n} |A(h' - h, \phi_i)|$.

After jointly training the reconstruction and selectivity losses, our algorithm disentangles four directed factors of variations as seen in Figure 6.4: $\pm x$-position and $\pm y$-position of the agent. For visualization purposes, in the rest of the section, we chose the bottleneck of the autoencoder to be of size $K = 2$.

The disentanglement appears clearly as the latent features corresponding to the x and y position are orthogonal in the latent space. Moreover, we notice that our algorithm assigns both actions `up` (white and pink dots in Figure 6.4.a) to the same feature. It also does not create a signifant mode for the feature corresponding to the action `down+left` (light blue dots in Figure 6.4.a) as this feature is already explained by features `down` and `left`.

As far as we know, such a clear recovery of x, y coordinates on a grid would not emerge without specific model-based objectives or regularizations using prior representation learning frameworks. In particular, traditional autoencoder-based frameworks would not be able to recover this structure (even allowing for some non-linear deformation) unless the specific structure of the grid could somehow be present in the inputs.



(a) (b)

Figure 6.4: (a) Sampling of 1000 variations $dh = h' - h$ and its kernel density estimation encountered when sampling random controllable factors ϕ. We observe that our algorithm disentangles these representations on 4 main modes, each corresponding to the action that was actually taken by the agent (pink and white for `up`, light blue for `down+left`, green for `right`, purple black `down` and night blue for `left`). (b) The disentangled stucture in the latent space. The x and y axis are disentangled such that we can recover the x and y position of the agent in any observation s simply by looking at its latent encoding $h = f(s)$. The missing point on this grid is the only position the agent cannot reach as it lies on an orange block.

6.3.3.2 Towards planning and policy inference

This disentangled structure could be used to address many challenging issues in reinforcement learning. We give two examples in figures 6.5 and 6.6:

- Model-based predictions: Given an initial state, s_0, and an action sequence $a_{\{0:T-1\}}$, we want to predict the resulting state s_T.
- A simplified deterministic policy inference problem: Given an initial state s_{start} and a terminal state s_{goal}, we aim to find a suitable action sequence $a_{\{0:T-1\}}$ such that s_{goal} can be reached from s_{start} by following it.

Because of the *tanh* activation on the last layer of $\phi(h, z)$, the different factors of variation $dh = h' - h$ are placed on the vertices of a hypercube of dimension K, and we can think of the the policy inference problem as finding a path in that simpler space, where the starting point is h_{start} and the goal is h_{goal}. We believe this could prove to be a much easier problem to solve.

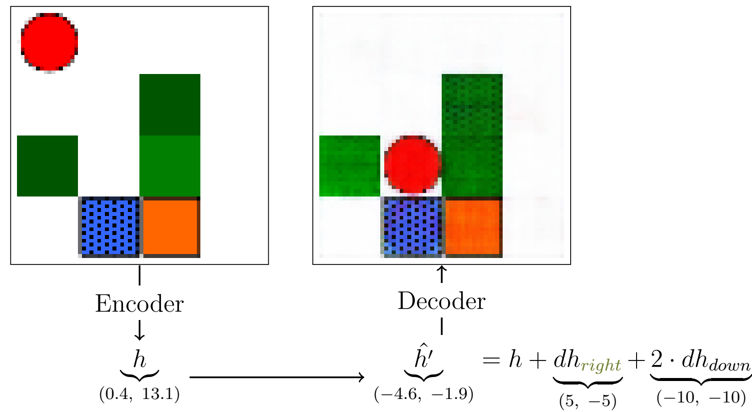


Figure 6.5: Predicting the effect of a cause on Mazebase. The leftmost image is the visual input of the environment, where the agent is the round circle, and the switch states are represented by shades of green. After the training, we are able to distinguish one cluster per dh (Figure 6.4), that is to say per variation obtained after performing an action, independently from the position h. Therefore, we are able to move the agent just by adding the corresponding dh to our latent representation h. The second image is just the reconstruction obtained by feeding the resulting h' into the decoder.

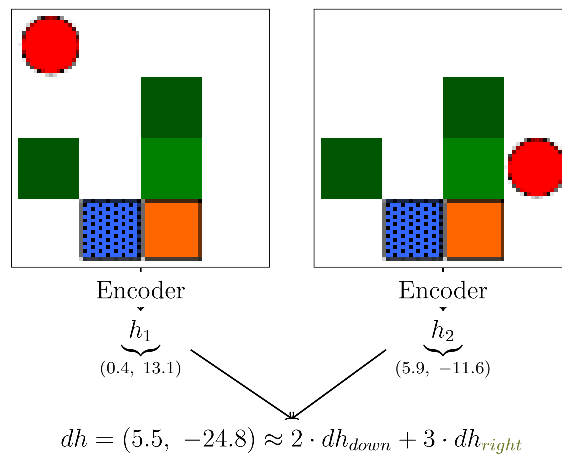


Figure 6.6: Given a starting state and a goal state, we are able to decompose the difference of the two representations dh into a (non-directed) sequence of movements.

However, this disentangled representation alone cannot solve completely these two issues in an arbitrary environment. Indeed, the only factors we are able to disentangle are the factors directly *controllable* by the agent, thus, we are not able to account for the ambiant dynamics or other agents' influence.

6.4 Discussion

We have introduced a novel method aiming at learning representations which disentangle the underlying factors of variation. Its main assumption is that some of those factors correspond to independently controllable aspects of the environment. This leads to training frameworks in which one learns jointly a set of exploratory policies and corresponding features of the learned representation which disentangle those controlled aspects.

6.4.1 Considering Instances and Object Diversity

In this work we focused on the simpler setups in which the environment is made of a static set of objects. In this case, if the objective posited in §6.2.1 is learned correctly, we can assume that feature k of the representation can unambiguously refer to some controllable property of some specific object in the environment. For example, the agent's world might contain only a red circle and a green rectangle, which are only affected by the actions of the agent (they do not move on their own) and we only change the positions and colours of these objects from one trial to the next. Hence, a specific feature f_k can learn to unambiguously refer to the position or the colour of one of these two objects.

In reality, environments are stochastic, and the set of objects in a given scene is drawn from some distribution. The number of objects may vary and their types may be different. It then becomes less obvious how feature k could refer in a clear way to some feature of one of the objects in a particular scene. If we have *instances* of objects of different types, some addressing or naming scheme is required to refer to the particular objects (instances) present in the scene, so as to match the policy with a particular attribute of a particular object to selectively modify. While our proposed distributed alternative (§6.2.2.1) is an attempt to address this, a fundamental representational problem remains.

This is connected to the binding problem in neuro-cognitive science: how to represent a set of objects, each having different attributes, so that we do not confuse, for example, the set {red circle, blue square} with {red square, blue circle}. The binding problem has received some attention in the representation learning literature (Minin et al., 2012; Greff et al., 2016), but still remains mostly unsolved. Jointly considering this problem and learning controllable features may prove fruitful.

These ideas may also lead to interesting ways of performing exploration. The RL exploration process could be driven by a notion of controllability, predicting the interestingness of objects in a scene and choosing features and associated policies with which to attempt controlling them – such ideas have only been briefly explored in the literature (e.g. Ratitch and Precup (2003)). How do humans choose with which object to play? We are attracted

to objects for which we do not yet know if and how we can control them, and such a process may be critical to learn how the world works.

6.4.2 The (Poor) Dynamics of Joint Disentanglement

An obvious extension of the proposed ideas is to consider that controllable aspects of an environment are only controllable over many time steps. Instead of considering that a controllable feature has an associated policy which *immediately* modifies it, we'd like to allow multiple time steps before a change in a feature.

While this is feasible, and indeed we have tried it experimentally, it leads to an interesting and possibly inevitable conundrum (also faced by options discovery, see e.g. Bacon et al., 2016). To put it simply, considering multiple time steps creates an exponentially large (in time) set of candidate "temporally-extended features" to choose from, jointly with a correspondingly large number of "temporally-extended actions" (commonly known as options).

While one could assume that our proposed independence constraint limits the potential subsets which end up as controllable features, in most environments there actually exists a virtually limitless number of pairs of features which satisfy mutual independence. Indeed practically, this problem is observed by agents choosing seemingly random sequences of actions as independently controllable features that modify pixel inputs in a cleverly (but useless) independent way.

This phenomenon is further compounded by the dynamics of learning policies *and* features at the same time. Since one depends on the other and vice-versa, a circular dependency emerges which easily prevents any sort of convergence of the parameters. While in the experimental results presented above this circular dependency could be resolved by tuning learning rates appropriately, this was because of the limited number of features and possible ways of partitioning the state space.

While more work needs to be done to understand this problem, and in general the problem of coordination between mutiple modules in deep neural network architectures (Bengio et al., 2016; Shazeer et al., 2017), it does seem possible that there are fundamental barriers here–barriers that cannot be crossed without including other fundamental inductive biases in temporal learning, such as affordances (Khetarpal et al., 2020) or causality (Goyal et al., 2020).

7

Synthesis

7.1 Summary of Contributions

What unifies the research in this thesis is a desire to understand fundamental phenomena behind the many methods used in Machine Learning. This knowledge not only propels us forward in the discovery of new and better techniques, it also offers us stable foundations to build upon. Indeed, much of science concerns building models of reality, but perhaps paradoxically, we now find ourselves building models of our models of reality. Deep neural networks having become so complex, we must now resort to understanding their function through high-level abstractions—models. Having the right abstractions, the right mental images, is thus crucial.

We started this thesis by providing such abstractions, and pointed to a more precise understanding of memorization and its function within neural networks as a mechanism for generalization. We then showed that analyzing interference reveals interesting patterns in neural networks, in particular when they are used to perform bootstrapping in Reinforcement Learning. These patterns help us understand why naive deep RL methods are sample inefficient and can appear to generalize poorly.

We then continued this thesis by taking to heart the insight that deep RL uses the wrong tools. Naively imported from the supervised deep learning literature, the assumptions that these tools rely on are broken in deep RL. We thus proposed a novel optimization method that takes into account the non-stationarity of the bootstrapping target in value prediction. We showed that this addresses the problem of staleness, which in turns helps learning value functions faster.

Out of a deeper comprehension of bootstrapping, we then proposed a novel framework for training generative models based on the notion of flow. By using bootstrapping to estimate flow we showed that we can efficiently generate large sets of diverse and high-reward candidates. Such a framework is shown to be effective in the context of drug discovery, but its possible applications seem to cast a much wider net. Indeed, in any black-box optimization problem where one cares about exploration and amortizing exploration over the course of training, our framework seems like a logical choice.

Finally, we ended the thesis by discussing work again related to generalization, joint representation learning of features and policies. We proposed a notion of independence and controllability which could be used to recover such objects, and showed that in simple environments the proposed method recovered the underlying factors that were under the agent's control.

7.2 Limitations

While the novel ideas presented in this thesis certainly move forward our comprehension of neural networks and deep reinforcement learning, they also point to a series of fundamental unknowns which remain to be understood, perhaps the most fundamental of which remains the question of generalization. How is it achieved? Are our current neural architectures enough to solve any generalization problem, or do we need new priors? New learning methods?

Even though we set out to understand generalization, it seems we are left with more questions than answers. In Chapter 3 we used to notion of interference to understand how learning algorithms such as Temporal Difference learning induce underparameterization and improper generalization in DNNs. Similar conclusions emerge from recent related work (Kumar et al., 2020), yet still do not offer a cohesive and actionable framework that helps us predict when generalization occurs.

Methodologically, Chapter 3 is also limited in the *kind* of generalization in measures, and only tackles fairly simple RL environments (Bellemare et al., 2013). Some much richer conceptualizations of generalization exist in RL (Whiteson et al., 2011); importantly, these suggest training and testing agents on different tasks and environments with structural commonalities, and have recently started to be adopted as good practice (Cobbe et al., 2020).

In Chapter 6, we also tackled generalization through priors, proposing novel ways of learning features which appeal to our intuitions about generalization. Yet, this intuition didn't prove to be enough, and we found that such ideas are missing the ingredients required for scale. What we learn from the exploration of those ideas is that the priors we choose to encode within our models can be powerful, but always have the danger of being too specific—too narrow to be widely useful, or too demanding to be easily learnable.

Another unknown related to generalization arises in Chapter 5. In this chapter we proposed a way to train generative models that behave in some desired ways. We opposed these models to MCMC methods that are reliant on iteration to produce samples. To do so, we claimed that we essentially amortize the iterative process that occurs in MCMC through

the training of our generative model. It is still unclear how and why this amortization happens, as we once again rely on the concept of generalization: assuming that generic patterns are learned means assuming that information is reused after each iteration of learning (as opposed to the iteration of the Markov chain in MCMC).

Relying on this assumption could be dangerous, since, at scale, measuring whether the "right" patterns are captured—or if only a simpler subspace of the generative domain is captured—is intractable. Instead we heavily rely on the generalization capabilities and assumptions provided by the deep learning literature. There are two dangers here currently known, the first is that deep architectures, while very capable of generalization within their training distribution, can fail in arbitrary ways outside of that distribution (Arjovsky, 2021). In our case, the model "chooses" its own distribution in some sense, so while it may appear to perform correctly around it, it may be missing important modes of the true distribution. The second danger is that of defining metrics of generalization, which has proven to be a challenge in both generative modelling (Heusel et al., 2017; Barratt and Sharma, 2018) and Reinforcement Learning (Kirk et al., 2021). What paradigm should be adopted in the context of the proposed method is still unclear.

Another important limitation of the work presented in Chapter 5, although perhaps more technical, is the lack of a sense of approximation error. Although we show that, at convergence, the models we learn provide us with a reward-proportional sampling mechanism, it's not clear how far off we become if the model is imperfect (which, in practice, it always is). While it could be that small errors in flows induce small errors in sampling probability, there could also be compounding effects, where small errors in flows lead to large errors in probability, emerging from the typically exponential number of paths combinatorially-generated objects have.

7.3 Future Work

One thing is certain from the last section, opportunities to keep digging and understanding the field of machine learning abound.

Yet, the discoveries in this thesis also led its author to a shifting perspective on the field: its existing tools are beginning to be strong enough to solve larger and larger real world problems, and so affect the lives of more and more people, hopefully, for the best.

This is probably best reflected in the motivating topic of Chapter 5, the problem of drug discovery and molecular design. Although computational methods have long been considered essential in this domain (McDonnell et al., 1995; Brown et al., 2004; Kawai et al., 2014), machine learning and deep learning methods now appear mature enough to

significantly contribute (Cao and Kipf, 2018; Popova et al., 2019; Swersky et al., 2020; Gottipati et al., 2020). Nonetheless, much remains to be done and understood: finding accurate models of uncertainty (Jain et al., 2021), understanding the generalization of architectures applied to molecules (Garg et al., 2020), finding efficient search methods in large combinatorial spaces (such as presented in Chapter 5), or predicting chemical properties (Gentile et al., 2020). All these challenges revolve around data and predictions, and so are challenges that machine learning can presumably help solve.

Going back to the specific method proposed in Chapter 5, two immediate next steps appear: first, developing strong theory around the methods would be valuable—approximation error bounds and the likes; second, developing more efficient or stable learning objectives. Even though interesting results were obtained by using the objective we propose, a number of stability issues very likely remain—issues intimately related to issues in Temporal Difference learning.

Indeed, the issues highlighted in Chapter 3 also suggest that a number of things could be improved in TD-based methods: architecture (Dong et al., 2020), objectives (e.g. TD(0) vs TD(λ), see §3.2.2.4), or optimizers (e.g. that of Chapter 4, see also Romoff et al. 2021).

While this thesis contains suggestions and possible solutions to the latter two problems, a number of opportunities remain. Novel objectives including auxiliary prediction tasks (Jaderberg et al., 2017) or self-supervision (Scholz et al., 2021) have been found to be very beneficial. In general, leveraging the data one already has to learn as much as possible from it appears very effective (Laskin et al., 2020). Understanding how these techniques improve generalization will be critical to improving them further. On the optimization side, existing methods that explicitly try to take the RL factor into account seem to struggle scaling up (Romoff et al., 2021; Bengio et al., 2020b), but the findings of Chapter 4 suggest that the fact that they work at all means some important differences exist between the stationary i.i.d. setting of supervised learning and the RL setting, which can potentially be leveraged to speed up learning. Adapting the supervised learning tools that we naively use in deep RL may reap interesting benefits beyond optimization (Li and Pathak, 2021).

The representation learning work of Chapter 6, in its failure to scale, also suggests that the right priors needed to learn useful and reusable abstractions remain elusive. More recent literature has many promising leads, affordances (Khetarpal et al., 2020), independent mechanisms (Goyal et al., 2020), architectures that plan implicitly (Schrittwieser et al., 2020), or temporally extended actions (Bagaria and Konidaris, 2019). The opportunities to take inspiration from biological intelligence are many, and can be tempting, even if slightly terminologically blurring the lines between the neuroscience and machine learning. For example, the use of concepts like *consciousness* (Zhao et al., 2021) or *dreaming* (Hafner

et al., 2019) have inspired real progress in the field.

~

We started this thesis with a discussion on cognitive tools and abstraction. If one thing is clear from only the last 5 years over which this thesis was written, deep learning has enabled the creation of models of ever-increasing complexity. While it may appear that the rate at which we create these new systems of abstraction outpaces the rate at which we're able to understand them, the emergence of a unifying theory seems to be on the horizon. More and more, the field is able to connect the dots between various phenomena occuring in neural networks (Frankle and Carbin, 2018; Li et al., 2018) and find useful predictive models of our own models (Kaplan et al., 2020). If we keep relentlessly going back to the fundamentals, and unifying all that which our vast field has learned, we may very well unlock the keys to cognition and intelligence.

Bibliography

A. Abid, M. Farooqi, and J. Zou. Persistent anti-muslim bias in large language models. *arXiv preprint arXiv:2101.05783*, 2021.

J. Achiam, E. Knight, and P. Abbeel. Towards characterizing divergence in deep q-learning. *arXiv preprint arXiv:1903.08894*, 2019.

R. Agarwal, D. Schuurmans, and M. Norouzi. Striving for simplicity in off-policy deep reinforcement learning. *arXiv preprint arXiv:1907.04543*, 2019.

M. Andrychowicz, A. Raichuk, P. Stańczyk, M. Orsini, S. Girgin, R. Marinier, L. Hussenot, M. Geist, O. Pietquin, M. Michalski, S. Gelly, and O. Bachem. What matters in on-policy reinforcement learning? a large-scale empirical study. *International Conference on Learning Representations*, 2020.

C. Angermueller, D. Dohan, D. Belanger, R. Deshpande, K. Murphy, and L. Colwell. Model-based reinforcement learning for biological sequence design. In *International Conference on Learning Representations*, 2020.

O. Anschel, N. Baram, and N. Shimkin. Averaged-dqn: Variance reduction and stabilization for deep reinforcement learning. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 176–185. JMLR. org, 2017.

H. Anton and C. Rorres. *Elementary linear algebra: applications version*. John Wiley & Sons, 2013.

M. Arjovsky. Out of distribution generalization in machine learning, 2021.

M. Arjovsky, L. Bottou, I. Gulrajani, and D. Lopez-Paz. Invariant risk minimization, 2019.

D. Arpit, S. Jastrzębski, N. Ballas, D. Krueger, E. Bengio, M. S. Kanwal, T. Maharaj, A. Fischer, A. Courville, Y. Bengio, et al. A closer look at memorization in deep networks. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 233–242. JMLR. org, 2017.

P.-L. Bacon, J. Harb, and D. Precup. The option-critic architecture. *Proceedings of the AAAI Conference on Artificial Intelligence*, 31, 2016.

A. Bagaria and G. Konidaris. Option discovery using deep skill chaining. In *International Conference on Learning Representations*, 2019.

D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate, 2014.

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

D. Bajusz, A. Rácz, and K. Héberger. Why is tanimoto index an appropriate choice for fingerprint-based similarity calculations? *Journal of cheminformatics*, 7(1):1–13, 2015.

D. Balduzzi, B. McWilliams, and T. Butler-Yeoman. Neural taylor approximations: Convergence and exploration in rectifier networks. In *International Conference on Machine Learning*, pages 351–360, 2017.

S. Barocas, M. Hardt, and A. Narayanan. Fairness in machine learning. *NeurIPS tutorial*, 1:2017, 2017.

S. Barratt and R. Sharma. A note on the inception score. *arXiv preprint arXiv:1801.01973*, 2018.

A. G. Barto, R. S. Sutton, and C. W. Anderson. Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE transactions on systems, man, and cybernetics*, (5):834–846, 1983.

J. Baxter and P. L. Bartlett. Infinite-horizon policy-gradient estimation. *Journal of Artificial Intelligence Research*, 15:319–350, 2001.

M. Belkin, D. Hsu, S. Ma, and S. Mandal. Reconciling modern machine learning practice and the bias-variance trade-off. *Proceedings of the National Academy of Sciences 116.32*, 2019.

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

M. G. Bellemare, W. Dabney, and R. Munos. A distributional perspective on reinforcement learning. *International Conference on Machine Learning*, 2017.

R. Bellman. Dynamic programming and lagrange multipliers. *Proceedings of the National Academy of Sciences*, 42(10):767–769, 1956.

R. Bellman. A markovian decision process. *Journal of mathematics and mechanics*, pages 679–684, 1957.

G. W. Bemis and M. A. Murcko. The properties of known drugs. 1. molecular frameworks. *Journal of medicinal chemistry*, 39(15):2887–2893, 1996.

E. Bengio, P.-L. Bacon, J. Pineau, and D. Precup. Conditional computation in neural networks for faster models. *Workshop Track: International Conference on Learning Representations*, 2016.

E. Bengio, V. Thomas, J. Pineau, D. Precup, and Y. Bengio. Independently controllable features, 2017.

E. Bengio, J. Pineau, and D. Precup. Interference and generalization in temporal difference learning. In *International Conference on Machine Learning*, 2020a.

E. Bengio, J. Pineau, and D. Precup. Correcting momentum in temporal difference learning. *arXiv preprint arXiv:2106.03955*, 2020b.

E. Bengio, M. Jain, M. Korablyov, D. Precup, and Y. Bengio. Flow network based generative models for non-iterative diverse candidate generation. *Advances in Neural Information Processing Systems*, 2021.

Y. Bengio, P. Lamblin, D. Popovici, and H. Larochelle. Greedy layer-wise training of deep networks. In *Advances in neural information processing systems*, pages 153–160, 2007.

Y. Bengio, A. Courville, and P. Vincent. Representation learning: A review and new perspectives. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 35(8): 1798–1828, Aug 2013a. ISSN 2160-9292. doi: 10.1109/tpami.2013.50.

Y. Bengio, A. Courville, and P. Vincent. Representation learning: A review and new perspectives. *IEEE transactions on pattern analysis and machine intelligence*, 35(8): 1798–1828, 2013b.

G. R. Bickerton, G. V. Paolini, J. Besnard, S. Muresan, and A. L. Hopkins. Quantifying the chemical beauty of drugs. *Nature chemistry*, 4(2):90–98, 2012.

A.-E. Birn. Philanthrocapitalism, past and present: The rockefeller foundation, the gates foundation, and the setting (s) of the international/global health agenda. *Hypothesis*, 12(1):e8, 2014.

C. M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006a. ISBN 0387310738.

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

A. L. Blum and R. L. Rivest. Training a 3-node neural network is np-complete. *Neural Networks*, 5(1):117–127, 1992.

A. Blumer, A. Ehrenfeucht, D. Haussler, and M. K. Warmuth. Occam's razor. *Information processing letters*, 24(6):377–380, 1987.

N. Bostrom. Existential risks: Analyzing human extinction scenarios and related hazards. *Journal of Evolution and technology*, 9, 2002.

N. Bostrom. *Superintelligence: Paths, dangers, strategies*. Oxford University Press, 2014.

A. Botev, G. Lever, and D. Barber. Nesterov's accelerated gradient and momentum as approximations to regularised update descent. In *2017 International Joint Conference on Neural Networks (IJCNN)*, pages 1899–1903. IEEE, 2017.

L. Bottou. Online learning and stochastic approximations. *On-line learning in neural networks*, 17(9):142, 1998.

D. Brandfonbrener and J. Bruna. Geometric insights into the convergence of nonlinear td learning. In *International Conference on Learning Representations*, 2020.

G. Brassard and P. Bratley. *Fundamentals of algorithmics*, volume 524. Prentice Hall Englewood Cliffs, 1996.

N. Brown, B. McKay, F. Gilardoni, and J. Gasteiger. A graph-based genetic algorithm and its application to the multiobjective evolution of median molecules. *Journal of chemical information and computer sciences*, 44(3):1079–1087, 2004.

T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

M. Brundage, S. Avin, J. Wang, H. Belfield, G. Krueger, G. Hadfield, H. Khlaaf, J. Yang, H. Toner, R. Fong, et al. Toward trustworthy ai development: mechanisms for supporting verifiable claims. *arXiv preprint arXiv:2004.07213*, 2020.

L. Buesing, N. Heess, and T. Weber. Approximate inference in discrete distributions with monte carlo tree search and value functions. *International Conference on Artificial Intelligence and Statistics*, 2019.

J. Buolamwini and T. Gebru. Gender shades: Intersectional accuracy disparities in commercial gender classification. In *Conference on fairness, accountability and transparency*, pages 77–91. PMLR, 2018.

N. D. Cao and T. Kipf. Molgan: An implicit generative model for small molecular graphs, 2018.

J. S. O. Ceron and P. S. Castro. Revisiting rainbow: Promoting more insightful and inclusive deep reinforcement learning research. In M. Meila and T. Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pages 1373–1383. PMLR, 18–24 Jul 2021.

M. Chen, A. Beutel, P. Covington, S. Jain, F. Belletti, and E. H. Chi. Top-k off-policy correction for a reinforce recommender system. In *Proceedings of the Twelfth ACM International Conference on Web Search and Data Mining*, pages 456–464, 2019.

T. Chen, S. Kornblith, M. Norouzi, and G. Hinton. A simple framework for contrastive learning of visual representations. *International Conference on Machine Learning*, 2020.

Y. Chen and B. Zhang. Learning to solve network flow problems via neural decoding. *arXiv preprint arXiv:2002.04091*, 2020.

A. Church. A note on the entscheidungsproblem. *The journal of symbolic logic*, 1(1):40–41, 1936.

K. Cobbe, O. Klimov, C. Hesse, T. Kim, and J. Schulman. Quantifying generalization in reinforcement learning. *Proceedings of the 36th International Conference on Machine Learning*, 97:1282–1289, 09–15 Jun 2019.

K. Cobbe, C. Hesse, J. Hilton, and J. Schulman. Leveraging procedural generation to benchmark reinforcement learning. In *International conference on machine learning*, pages 2048–2056. PMLR, 2020.

J. Cohen. *Statistical power analysis for the behavioral sciences*. Routledge, 2013.

G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314, 1989.

H. Dai, R. Singh, B. Dai, C. Sutton, and D. Schuurmans. Learning discrete energy-based models via auxiliary-variable local exploration. In *Neural Information Processing Systems (NeurIPS)*, 2020.

F. Dangel, F. Kunstner, and P. Hennig. BackPACK: Packing more into backprop. In *International Conference on Learning Representations*, 2020.

Y. N. Dauphin, R. Pascanu, C. Gulcehre, K. Cho, S. Ganguli, and Y. Bengio. Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. In *Advances in neural information processing systems*, pages 2933–2941, 2014.

P. Dayan. Improving generalization for temporal difference learning: The successor representation. *Neural Computation*, 5(4):613–624, 1993.

P. Dayan, G. E. Hinton, R. M. Neal, and R. S. Zemel. The Helmholtz machine. *Neural computation*, 7(5):889–904, 1995.

A. de Brébisson and P. Vincent. An exploration of softmax alternatives belonging to the spherical loss family. *International Conference on Learning Representations*, 2015.

J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

J. Diamond. *Guns, Germs, and Steel*. WW Norton Publishing, 1997.

L. Dinh, D. Krueger, and Y. Bengio. NICE: Non-linear Independent Components Estimation. arXiv:1410.8516, ICLR 2015 workshop, 2014.

K. Dong, Y. Luo, T. Yu, C. Finn, and T. Ma. On the expressivity of neural networks for deep reinforcement learning. In *International Conference on Machine Learning*, pages 2627–2637. PMLR, 2020.

F. Draxler, K. Veschgini, M. Salmhofer, and F. A. Hamprecht. Essentially no barriers in neural network energy landscape. *Proceedings of the 35th International Conference on Machine Learning*, 2018.

G. K. Dziugaite, A. Drouin, B. Neal, N. Rajkumar, E. Caballero, L. Wang, I. Mitliagkas, and D. M. Roy. In search of robust measures of generalization. *Advances in Neural Information Processing Systems*, 33, 2020.

A. P. Engelbrecht. Using the taylor expansion of multilayer feedforward neural networks. *South African Computer Journal*, 2000(26):181–189, 2000.

A. P. Engelbrecht. A new pruning heuristic based on variance analysis of sensitivity information. *IEEE Transactions on Neural Networks*, 12(6):1386–1399, 2001.

J. Farebrother, M. C. Machado, and M. Bowling. Generalization and regularization in dqn. *arXiv preprint arXiv:1810.00123*, 2018.

W. Fedus, P. Ramachandran, R. Agarwal, Y. Bengio, H. Larochelle, M. Rowland, and W. Dabney. Revisiting fundamentals of experience replay. *International Conference on Machine Learning. PMLR*, 2020.

T. S. Ferguson. A bayesian analysis of some nonparametric problems. *The annals of statistics*, pages 209–230, 1973.

M. Fisher. *Capitalist realism: Is there no alternative?* John Hunt Publishing, 2009.

P. Foret, A. Kleiner, H. Mobahi, and B. Neyshabur. Sharpness-aware minimization for efficiently improving generalization. *International Conference on Learning Representations*, 2021.

E. W. Forgy. Cluster analysis of multivariate data: efficiency versus interpretability of classifications. *biometrics*, 21:768–769, 1965.

S. Fort and S. Ganguli. Emergent properties of the local geometry of neural loss landscapes. *CoRR*, 2019.

S. Fort, P. K. Nowak, S. Jastrzebski, and S. Narayanan. Stiffness: A new perspective on generalization in neural networks, 2019.

J. Frankle and M. Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks. In *International Conference on Learning Representations*, 2018.

K. Fukushima. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological cybernetics*, 36(4):193–202, 1980.

V. Garg, S. Jegelka, and T. Jaakkola. Generalization and representational limits of graph neural networks. In H. D. III and A. Singh, editors, *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 3419–3430. PMLR, 13–18 Jul 2020.

T. Garipov, P. Izmailov, D. Podoprikhin, D. P. Vetrov, and A. G. Wilson. Loss surfaces, mode connectivity, and fast ensembling of dnns. In *NeurIPS*, 2018.

F. Gentile, V. Agrawal, M. Hsing, A.-T. Ton, F. Ban, U. Norinder, M. E. Gleave, and A. Cherkasov. Deep docking: a deep learning platform for augmentation of structure based drug discovery. *ACS central science*, 6(6):939–949, 2020.

M. Gilens and B. I. Page. Testing theories of american politics: Elites, interest groups, and average citizens. *Perspectives on politics*, 12(3):564–581, 2014.

J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl. Neural message passing for quantum chemistry. *International Conference on Machine Learning*, 2017.

C. Gini. Variabilità e mutabilità. *Reprinted in Memorie di metodologica statistica (Ed. Pizetti E*, 1912.

X. Glorot, A. Bordes, and Y. Bengio. Deep sparse rectifier neural networks. In *Aistats*, volume 15, page 275, 2011.

P. W. Glynn. Likelilood ratio gradient estimation: an overview. In *Proceedings of the 19th conference on Winter simulation*, pages 366–375. ACM, 1987.

I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. Generative Adversarial Networks. In *NIPS'2014*, 2014.

M. Gori and A. Tesi. On the problem of local minima in backpropagation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 14(1):76–86, 1992.

S. K. Gottipati, B. Sattarov, S. Niu, Y. Pathak, H. Wei, S. Liu, K. M. J. Thomas, S. Blackburn, C. W. Coley, J. Tang, S. Chandar, and Y. Bengio. Learning to navigate the synthetically accessible chemical space using reinforcement learning. *International Conference on Machine Learning*, 2020.

A. Goyal, A. Lamb, J. Hoffmann, S. Sodhani, S. Levine, Y. Bengio, and B. Schölkopf. Recurrent independent mechanisms. *International Conference on Learning Representations*, 2020.

W. Grathwohl, K. Swersky, M. Hashemi, D. Duvenaud, and C. J. Maddison. Oops i took a gradient: Scalable sampling for discrete distributions. *International Conference on Machine Learning*, 2021.

A. Graves. Generating sequences with recurrent neural networks, 2013.

K. Greff, A. Rasmus, M. Berglund, T. Hao, H. Valpola, and J. Schmidhuber. Tagger: Deep unsupervised perceptual grouping. In *Advances in Neural Information Processing Systems*, pages 4484–4492, 2016.

D. Gupta. Applicability of momentum in the methods of temporal learning, 2020.

H. Gupta, R. Srikant, and L. Ying. Finite-time performance bounds and adaptive learning rate selection for two time-scale reinforcement learning. In *Advances in Neural Information Processing Systems*, pages 4704–4713, 2019.

T. Haarnoja, H. Tang, P. Abbeel, and S. Levine. Reinforcement learning with deep energy-based policies. In *International Conference on Machine Learning*, pages 1352–1361. PMLR, 2017.

T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor, 2018.

R. Hadsell, S. Chopra, and Y. LeCun. Dimensionality reduction by learning an invariant mapping. In *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'06)*, volume 2, pages 1735–1742. IEEE, 2006.

D. Hafner, T. Lillicrap, J. Ba, and M. Norouzi. Dream to control: Learning behaviors by latent imagination. *International Conference on Learning Representations*, 2019.

Y. N. Harari. *Sapiens: A brief history of humankind*. Random House, 2014.

M. Hardt, B. Recht, and Y. Singer. Train faster, generalize better: Stability of stochastic gradient descent. In *International Conference on Machine Learning*, pages 1225–1234, 2016.

B. Hassibi and D. G. Stork. Second order derivatives for network pruning: Optimal brain surgeon. In *Advances in neural information processing systems*, pages 164–171, 1993.

W. K. Hastings. Monte carlo sampling methods using markov chains and their applications. *Biometrika*, 57, 1970.

K. He, X. Zhang, S. Ren, and J. Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *arXiv preprint arXiv:1502.01852*, 2015.

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

M. Hessel, J. Modayil, H. Van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. Azar, and D. Silver. Rainbow: Combining improvements in deep reinforcement learning. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

M. Heusel, H. Ramsauer, T. Unterthiner, B. Nessler, and S. Hochreiter. Gans trained by a two time-scale update rule converge to a local nash equilibrium. *Advances in neural information processing systems*, 30, 2017.

G. Hinton, N. Srivastava, and K. Swersky. Neural networks for machine learning lecture 6a overview of mini-batch gradient descent. *CSC321*, 2012.

G. E. Hinton and R. R. Salakhutdinov. Reducing the dimensionality of data with neural networks. *science*, 313(5786):504–507, 2006.

G. E. Hinton, S. Osindero, and Y.-W. Teh. A fast learning algorithm for deep belief nets. *Neural computation*, 18(7):1527–1554, 2006.

S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8): 1735–1780, 1997.

S. Hooker, N. Moorosi, G. Clark, S. Bengio, and E. Denton. Characterising bias in compressed models, 2020.

K. Hornik. Approximation capabilities of multilayer feedforward networks. *Neural Networks*, 4(2):251 – 257, 1991. ISSN 0893-6080.

E. Hutchins. *Cognition in the Wild*. MIT press, 1995.

A. Hyvarinen and H. Morioka. Unsupervised Feature Extraction by Time-Contrastive Learning and Nonlinear ICA. In *NIPS*, 2016.

M. Igl, G. Farquhar, J. Luketina, W. Boehmer, and S. Whiteson. The impact of non-stationarity on generalisation in deep reinforcement learning, 2020.

A. Ilyas, L. Engstrom, S. Santurkar, D. Tsipras, F. Janoos, L. Rudolph, and A. Madry. Are deep policy gradient algorithms truly policy gradient algorithms? *CoRR*, 2018.

S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

A. Jacot, F. Gabriel, and C. Hongler. Neural tangent kernel: Convergence and generalization in neural networks. In *Advances in neural information processing systems*, pages 8571–8580, 2018.

M. Jaderberg, V. Mnih, W. M. Czarnecki, T. Schaul, J. Z. Leibo, D. Silver, and K. Kavukcuoglu. Reinforcement learning with unsupervised auxiliary tasks. *International Conference on Learning Representations*, 2017.

M. Jain, S. Lahlou, H. Nekoei, V. Butoi, P. Bertin, J. Rector-Brooks, M. Korablyov, and Y. Bengio. Deup: Direct epistemic uncertainty prediction. *CoRR*, 2021.

J. H. Jensen. A graph-based genetic algorithm and generative model/monte carlo tree search for the exploration of chemical space. *Chemical science*, 10(12):3567–3572, 2019.

W. Jin, R. Barzilay, and T. Jaakkola. Chapter 11. junction tree variational autoencoder for molecular graph generation. *Drug Discovery*, page 228–249, 2020. ISSN 2041-3211.

N. Justesen, R. Rodriguez Torrado, P. Bontrager, A. Khalifa, J. Togelius, and S. Risi. Illuminating generalization in deep reinforcement learning through procedural level generation. In *NeurIPS Workshop on Deep Reinforcement Learning*, 2018.

C. Jutten and J. Herault. Blind separation of sources, part i: An adaptive algorithm based on neuromimetic architecture. *Signal processing*, 24(1):1–10, 1991.

D. Kahneman. *Thinking, fast and slow.* Macmillan, 2011.

J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, and D. Amodei. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020.

S. Kapturowski, G. Ostrovski, W. Dabney, J. Quan, and R. Munos. Recurrent experience replay in distributed reinforcement learning. In *International Conference on Learning Representations*, 2019.

K. Kawai, N. Nagata, and Y. Takahashi. De novo design of drug-like molecules by a fragment-based molecular evolutionary approach. *Journal of chemical information and modeling*, 54(1):49–56, 2014.

N. S. Keskar, D. Mudigere, J. Nocedal, M. Smelyanskiy, and P. T. P. Tang. On large-batch training for deep learning: Generalization gap and sharp minima. *International Conference on Learning Representations*, 2017a.

N. S. Keskar, D. Mudigere, J. Nocedal, M. Smelyanskiy, and P. T. P. Tang. On large-batch training for deep learning: Generalization gap and sharp minima. *5th International Conference on Learning Representations, ICLR*, 2017b.

K. Khetarpal, Z. Ahmed, G. Comanici, D. Abel, and D. Precup. What can i do here? a theory of affordances in reinforcement learning. In *International Conference on Machine Learning*, pages 5243–5253. PMLR, 2020.

D. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

D. Kingma and J. Ba. Adam: a method for stochastic optimization (2014). *International Conference on Learning Representations*, 15, 2015.

D. P. Kingma and M. Welling. Auto-encoding variational Bayes. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2014.

R. Kirk, A. Zhang, E. Grefenstette, and T. Rocktäschel. A survey of generalisation in deep reinforcement learning, 2021.

A. Kirsch, J. van Amersfoort, and Y. Gal. Batchbald: Efficient and diverse batch acquisition for deep bayesian active learning. *Advances in neural information processing systems*, 2019.

B. Krishnapuram, L. Carin, M. A. Figueiredo, and A. J. Hartemink. Sparse multinomial logistic regression: Fast algorithms and generalization bounds. *IEEE transactions on pattern analysis and machine intelligence*, 27(6):957–968, 2005.

A. Krizhevsky and G. Hinton. Learning multiple layers of features from tiny images, 2009.

A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

T. D. Kulkarni, A. Saeedi, S. Gautam, and S. J. Gershman. Deep successor reinforcement learning. *arXiv preprint arXiv:1606.02396*, 2016.

A. Kumar, A. Voet, and K. Zhang. Fragment based drug design: from experimental to computational approaches. *Current medicinal chemistry*, 19(30):5128–5147, 2012.

A. Kumar, R. Agarwal, D. Ghosh, and S. Levine. Implicit under-parameterization inhibits data-efficient deep reinforcement learning, 2020.

C. L. Lan, S. Tu, A. Oberman, R. Agarwal, and M. G. Bellemare. On the generalization of representations in reinforcement learning. *arXiv preprint arXiv:2203.00543*, 2022.

G. Landrum. Rdkit: Open-source cheminformatics. URL http://www.rdkit.org.

M. Laskin, K. Lee, A. Stooke, L. Pinto, P. Abbeel, and A. Srinivas. Reinforcement learning with augmented data. *Advances in Neural Information Processing Systems*, 33, 2020.

Y. LeCun. The MNIST database of handwritten digits. *http://yann.lecun.com/exdb/mnist/*, 1998.

Y. LeCun, J. S. Denker, and S. A. Solla. Optimal brain damage. In *Advances in neural information processing systems*, pages 598–605, 1990.

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

Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

J. Leike, D. Krueger, T. Everitt, M. Martic, V. Maini, and S. Legg. Scalable agent alignment via reward modeling: a research direction, 2018.

A. Li and D. Pathak. Functional regularization for reinforcement learning via learned fourier features. *Advances in Neural Information Processing Systems*, 34, 2021.

C. Li, H. Farkhoor, R. Liu, and J. Yosinski. Measuring the intrinsic dimension of objective landscapes. In *International Conference on Learning Representations*, 2018.

T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra. Continuous control with deep reinforcement learning, 2016.

V. Liu, R. Kumaraswamy, L. Le, and M. White. The utility of sparse representations for control in reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 4384–4391, 2019a.

V. Liu, H. Yao, and M. White. Toward understanding catastrophic interference invalue-based reinforcement learning. *NeurIPS 2019 Optimization Foundations for Reinforcement Learning Workshop*, 2019b.

S. P. Lloyd. Least squares quantization in pcm. *IEEE Transactions on Information Theory*, 28:129–137, 1982.

D. Lopez-Paz and M. Ranzato. Gradient episodic memory for continual learning. In *Advances in Neural Information Processing Systems*, pages 6467–6476, 2017.

Y. Luo, K. Yan, and S. Ji. Graphdf: A discrete flow model for molecular graph generation, 2021.

A. L. Maas, A. Y. Hannun, and A. Y. Ng. Rectifier nonlinearities improve neural network acoustic models. In *Proceedings of the International Conference on Machine learning*, 2013.

M. C. Machado, M. G. Bellemare, E. Talvitie, J. Veness, M. Hausknecht, and M. Bowling. Revisiting the arcade learning environment: Evaluation protocols and open problems for general agents. *Journal of Artificial Intelligence Research*, 61:523–562, 2018.

D. Maclaurin, D. Duvenaud, and R. P. Adams. Gradient-based hyperparameter optimization through reversible learning. *International Conference on Machine Learning*, 2015.

W. S. McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943.

J. R. McDonnell, R. G. Reynolds, and D. B. Fogel. Docking conformationally flexible small molecules into a protein binding site through evolutionary programming. *Proceedings of the Fourth Annual Conference on Evolutionary Programming*, 1995.

N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller. Equation of state calculations by fast computing machines. *The journal of chemical physics*, 21 (6):1087–1092, 1953.

D. Meyer, R. Degenne, A. Omrane, and H. Shen. Accelerated gradient temporal difference learning algorithms. In *2014 IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning (ADPRL)*, pages 1–8. IEEE, 2014.

A. Minin, A. Knoll, H.-G. Zimmermann, A. Siemens, and L. Siemens. Complex Valued Artificial Recurrent Neural Network as a Novel Approach to Model the Perceptual Binding Problem. In *ESANN*. Citeseer, 2012.

M. Minsky. Steps toward artificial intelligence. *Proceedings of the IRE*, 49(1):8–30, 1961.

V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937, 2016.

P. Molchanov, S. Tyree, T. Karras, T. Aila, and J. Kautz. Pruning convolutional neural networks for resource efficient inference. *International Conference on Learning Representations*, 2016.

G. F. Montufar, R. Pascanu, K. Cho, and Y. Bengio. On the number of linear regions of deep neural networks. In *Advances in neural information processing systems*, pages 2924–2932, 2014.

A. Morcos, D. G. Barrett, N. C. Rabinowitz, and M. Botvinick. On the importance of single directions for generalization. *International Conference on Learning Representations*, 2018.

R. Munos, T. Stepleton, A. Harutyunyan, and M. Bellemare. Safe and efficient off-policy reinforcement learning. In *Advances in Neural Information Processing Systems*, pages 1054–1062, 2016.

K. P. Murphy. *Machine learning: a probabilistic perspective*. MIT press, 2012.

P. Nakkiran, G. Kaplun, Y. Bansal, T. Yang, B. Barak, and I. Sutskever. Deep double descent: Where bigger models and more data hurt. *International Conference on Learning Representations*, 2019.

C. Nash and C. Durkan. Autoregressive energy machines. In *International Conference on Machine Learning*, pages 1735–1744. PMLR, 2019.

A. Nazemi and F. Omidi. A capable neural network model for solving the maximum flow problem. *Journal of Computational and Applied Mathematics*, 236(14):3498–3513, 2012.

B. Neal, S. Mittal, A. Baratin, V. Tantia, M. Scicluna, S. Lacoste-Julien, and I. Mitliagkas. A modern take on the bias-variance tradeoff in neural networks, 2019.

D. M. Negoescu, P. I. Frazier, and W. B. Powell. The knowledge-gradient algorithm for sequencing experiments in drug discovery. *INFORMS Journal on Computing*, 23(3): 346–363, 2011. ISSN 1526-5528.

Y. Nesterov. A method of solving a convex programming problem with convergence rate $o(1/k^2)$ o (1/k2). In *Sov. Math. Dokl*, volume 27, 1983.

Y. Netzer, T. Wang, A. Coates, A. Bissacco, B. Wu, and A. Y. Ng. Reading digits in natural images with unsupervised feature learning. *NIPS Workshop on Deep Learning and Unsupervised Feature Learning*, 2011.

A. Y. Ng, S. J. Russell, et al. Algorithms for inverse reinforcement learning. In *Icml*, volume 1, page 2, 2000.

A. Nichol, J. Achiam, and J. Schulman. On first-order meta-learning algorithms. *arXiv preprint arXiv:1803.02999*, 2018.

S. U. Noble. *Algorithms of oppression*. New York University Press, 2018.

J. Oh, X. Guo, H. Lee, R. L. Lewis, and S. Singh. Action-conditional video prediction using deep networks in atari games. In *Advances in Neural Information Processing Systems*, pages 2863–2871, 2015.

A. v. d. Oord, S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. Senior, and K. Kavukcuoglu. Wavenet: A generative model for raw audio. *arXiv preprint arXiv:1609.03499*, 2016.

S. Oymak, Z. Fabian, M. Li, and M. Soltanolkotabi. Generalization guarantees for neural networks via harnessing the low-rank structure of the jacobian. *arXiv preprint arXiv:1906.05392*, 2019.

R. K. Pace and R. Barry. Sparse spatial autoregressions. *Statistics & Probability Letters*, 33(3):291–297, 1997. ISSN 0167-7152.

C. Packer, K. Gao, J. Kos, P. Krähenbühl, V. Koltun, and D. Song. Assessing generalization in deep reinforcement learning. *arXiv preprint arXiv:1810.12282*, 2018.

Y. Pan, A. White, and M. White. Accelerated gradient temporal difference learning. *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

G. Panuccio, A. Guez, R. Vincent, M. Avoli, and J. Pineau. Adaptive control of epileptiform excitability in an in vitro model of limbic seizures. *Experimental neurology*, 241:179–183, 2013.

A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019a.

A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in neural information processing systems*, pages 8026–8037, 2019b.

K. Pearson. Liii. on lines and planes of closest fit to systems of points in space. *The London, Edinburgh, and Dublin philosophical magazine and journal of science*, 2(11): 559–572, 1901.

A. Pektaş and T. Acarman. Deep learning to detect botnet via network flow summaries. *Neural Computing and Applications*, 31(11):8021–8033, 2019.

H. Penedones, D. Vincent, H. Maennel, S. Gelly, T. Mann, and A. Barreto. Temporal difference learning with neural networks-study of the leakage propagation problem. *arXiv preprint arXiv:1807.03064*, 2018.

C. C. Perez. *Invisible women: Exposing data bias in a world designed for men*. Random House, 2019.

. Pew Research Center. In their own words: Behind americans' views of 'socialism' and 'capitalism', 2019.

T. Piketty. *Capital in the twenty-first century*. Harvard University Press, 2018.

L. Pineda, S. Basu, A. Romero, R. Calandra, and M. Drozdzal. Active mr k-space sampling with reinforcement learning. In *International Conference on Medical Image Computing and Computer-Assisted Intervention*, pages 23–33. Springer, 2020.

B. T. Polyak. Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics*, 4(5):1–17, 1964.

M. Popova, M. Shvets, J. Oliva, and O. Isayev. Molecularrnn: Generating realistic molecular graphs with optimized properties, 2019.

D. Precup. *Temporal abstraction in reinforcement learning*. PhD thesis, University of Massachusetts Amherst, 2000.

M. L. Puterman. Markov decision processes: Discrete stochastic dynamic programming, 1994.

M. Raghu, J. Gilmer, J. Yosinski, and J. Sohl-Dickstein. Svcca: Singular vector canonical correlation analysis for deep learning dynamics and interpretability. In *Advances in Neural Information Processing Systems*, pages 6076–6085, 2017.

R. Rahul, T. Anjali, V. K. Menon, and K. Soman. Deep learning for network flow analysis and malware classification. In *International Symposium on Security in Computing and Communication*, pages 226–235. Springer, 2017.

M. Ranzato, A. Szlam, J. Bruna, M. Mathieu, R. Collobert, and S. Chopra. Video (language) modeling: a baseline for generative models of natural videos. *CoRR*, 2014.

C. E. Rasmussen et al. The infinite gaussian mixture model. In *NIPS*, volume 12, pages 554–560. Citeseer, 1999.

B. Ratitch and D. Precup. Using MDP Characteristics to Guide Exploration in Reinforcement Learning. In *ECML*, pages 313–324, 2003.

M. H. Ribeiro, R. Ottoni, R. West, V. A. Almeida, and W. Meira Jr. Auditing radicalization pathways on youtube. In *Proceedings of the 2020 conference on fairness, accountability, and transparency*, pages 131–141, 2020.

M. Riemer, I. Cases, R. Ajemian, M. Liu, I. Rish, Y. Tu, and G. Tesauro. Learning to learn without forgetting by maximizing transfer and minimizing interference. *International Conference on Learning Representations*, 2018.

S. Rifai, G. Mesnil, P. Vincent, X. Muller, Y. Bengio, Y. Dauphin, and X. Glorot. Higher order contractive auto-encoder. In *Joint European conference on machine learning and knowledge discovery in databases*, pages 645–660. Springer, 2011.

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

C. Robert and G. Casella. Monte carlo statistical methods. In *Springer Texts in Statistics*, 2004.

J. Romoff, P. Henderson, D. Kanaa, E. Bengio, A. Touati, P.-L. Bacon, and J. Pineau. Tdprop: Does jacobi preconditioning help temporal difference learning? *Proceedings of the 20th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2021)*, 2021.

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

D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *Cognitive modeling*, 5, 1988.

S. Russell and P. Norvig. *Artificial intelligence: a modern approach.* Prentice Hall, 2002.

A. E. Sallab, M. Abdou, E. Perot, and S. Yogamani. Deep reinforcement learning framework for autonomous driving. *Electronic Imaging*, 2017(19):70–76, 2017.

P. Samuelson and W. Nordhaus. *Economics*, chapter 1 (section) Market, Command, and Mixed Economies. McGraw-Hill, New York, 2007.

M. Sarigül and M. Avci. Performance comparison of different momentum techniques on deep reinforcement learning. *Journal of Information and Telecommunication*, 2(2):205–216, 2018. doi: 10.1080/24751839.2018.1440453.

K. Scaman and A. Virmaux. Lipschitz regularity of deep neural networks: analysis and efficient estimation, 2018.

R. E. Schapire and M. K. Warmuth. On the worst-case analysis of temporal-difference learning algorithms. *Machine Learning*, 22(1-3):95–121, 1996.

T. Schaul, J. Quan, I. Antonoglou, and D. Silver. Prioritized experience replay. *International Conference on Learning Representations*, 2015.

T. Schaul, D. Borsa, J. Modayil, and R. Pascanu. Ray interference: a source of plateaus in deep reinforcement learning. *CoRR*, 2019.

J. Schmidhuber. A possibility for implementing curiosity and boredom in model-building neural controllers. In *Proc. of the international conference on simulation of adaptive behavior: From animals to animats*, pages 222–227, 1991.

J. Scholz, C. Weber, M. B. Hafez, and S. Wermter. Improving model-based reinforcement learning with internal state representations through self-supervision. *arXiv preprint arXiv:2102.05599*, 2021.

J. Schrittwieser, I. Antonoglou, T. Hubert, K. Simonyan, L. Sifre, S. Schmitt, A. Guez, E. Lockhart, D. Hassabis, T. Graepel, et al. Mastering atari, go, chess and shogi by planning with a learned model. *Nature*, 588(7839):604–609, 2020.

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

A. Seff, W. Zhou, F. Damani, A. Doyle, and R. P. Adams. Discrete object generation with reversible inductive construction. *Advances in neural information processing systems*, 2019.

C. E. Shannon. A mathematical theory of communication. *The Bell system technical journal*, 27(3):379–423, 1948.

N. Shazeer, A. Mirhoseini, K. Maziarz, A. Davis, Q. Le, G. Hinton, and J. Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *International Conference on Learning Representations*, 2017.

C. Shi, M. Xu, Z. Zhu, W. Zhang, M. Zhang, and J. Tang. Graphaf: a flow-based autoregressive model for molecular graph generation. *International Conference on Learning Representations*, 2020.

Y. Shi, Z. Huang, S. Feng, H. Zhong, W. Wang, and Y. Sun. Masked label prediction: Unified message passing model for semi-supervised classification, 2021.

D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller. Deterministic policy gradient algorithms. In *Proceedings of the 31th International Conference on Machine Learning, ICML 2014, Beijing, China, 21-26 June 2014*, pages 387–395, 2014.

D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, et al. Mastering the game of go without human knowledge. *nature*, 550(7676):354–359, 2017.

D. Silver, S. Singh, D. Precup, and R. S. Sutton. Reward is enough. *Artificial Intelligence*, page 103535, 2021.

S. Singh, A. G. Barto, and N. Chentanez. Intrinsically motivated reinforcement learning. Technical report, MASSACHUSETTS UNIV AMHERST DEPT OF COMPUTER SCIENCE, 2005.

S. P. Singh and R. S. Sutton. Reinforcement learning with replacing eligibility traces. *Machine learning*, 22(1-3):123–158, 1996.

B. F. Skinner. *Science and human behavior*. Simon and Schuster, 1953.

N. Srinivas, A. Krause, S. Kakade, and M. Seeger. Gaussian process optimization in the bandit setting: No regret and experimental design. In *ICML*, 2010.

T. Sterling and J. J. Irwin. Zinc 15–ligand discovery for everyone. *Journal of chemical information and modeling*, 55(11):2324–2337, 2015.

J. Stewart. *Calculus: Concepts and contexts*. Cengage Learning, 2009.

S. Still and D. Precup. An information-theoretic approach to curiosity-driven reinforcement learning. *Theory in Biosciences*, 131(3):139–148, 2012.

S. Sukhbaatar, A. Szlam, G. Synnaeve, S. Chintala, and R. Fergus. MazeBase: A sandbox for learning from games. *arXiv preprint arXiv:1511.07401*, 2015.

T. Sun, H. Shen, T. Chen, and D. Li. Adaptive temporal difference learning with linear function approximation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2021.

I. Sutskever, J. Martens, G. Dahl, and G. Hinton. On the importance of initialization and momentum in deep learning. In *International conference on machine learning*, pages 1139–1147, 2013.

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

R. S. Sutton. Generalization in reinforcement learning: Successful examples using sparse coarse coding. In *Advances in neural information processing systems*, pages 1038–1044, 1996.

R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

R. S. Sutton, D. A. McAllester, S. P. Singh, Y. Mansour, et al. Policy gradient methods for reinforcement learning with function approximation. In *NIPS*, volume 99, pages 1057–1063, 1999a.

R. S. Sutton, D. Precup, and S. Singh. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial intelligence*, 112(1-2):181–211, 1999b.

R. S. Sutton, C. Szepesvári, and H. R. Maei. A convergent o (n) algorithm for off-policy temporal-difference learning with linear function approximation. *Advances in neural information processing systems*, 21(21):1609–1616, 2008.

R. S. Sutton, H. R. Maei, D. Precup, S. Bhatnagar, D. Silver, C. Szepesvári, and E. Wiewiora. Fast gradient-descent methods for temporal-difference learning with linear function approximation. In A. P. Danyluk, L. Bottou, and M. L. Littman, editors, *ICML*, volume 382 of *ACM International Conference Proceeding Series*, page 125. ACM, 2009a. ISBN 978-1-60558-516-1.

R. S. Sutton, H. R. Maei, D. Precup, S. Bhatnagar, D. Silver, C. Szepesvári, and E. Wiewiora. Fast gradient-descent methods for temporal-difference learning with linear function approximation. In *Proceedings of the 26th Annual International Conference on Machine Learning*, pages 993–1000, 2009b.

K. Swersky, Y. Rubanova, D. Dohan, and K. Murphy. Amortized bayesian optimization over discrete spaces. In *Conference on Uncertainty in Artificial Intelligence*, pages 769–778. PMLR, 2020.

C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus. Intriguing properties of neural networks. *International Conference on Learning Representations*, 2014.

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

P. Thodoroff, A. Durand, J. Pineau, and D. Precup. Temporal regularization for markov decision process. In *Advances in Neural Information Processing Systems*, pages 1779–1789, 2018.

V. Thomas, J. Pondard, E. Bengio, M. Sarfati, P. Beaudoin, M.-J. Meurs, J. Pineau, D. Precup, and Y. Bengio. Independently controllable factors, 2017.

V. Thomas, E. Bengio, W. Fedus, J. Pondard, P. Beaudoin, H. Larochelle, J. Pineau, D. Precup, and Y. Bengio. Disentangling the independently controllable factors of variation by interacting with the world. *arXiv preprint arXiv:1802.09484*, 2018.

T. Tieleman and G. Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural Networks for Machine Learning*, 4:2, 2012.

O. Trott and A. J. Olson. Autodock vina: improving the speed and accuracy of docking with a new scoring function, efficient optimization, and multithreading. *Journal of computational chemistry*, 31(2):455–461, 2010.

A. M. Turing. Computing machinery and intelligence. In *Parsing the turing test*, pages 23–65. Springer, 1950.

A. M. Turing et al. On computable numbers, with an application to the entscheidungsproblem. *J. of Math*, 58(345-363):5, 1936.

H. van Hasselt, A. Guez, and D. Silver. Deep reinforcement learning with double q-learning. *Proceedings of the AAAI conference on artificial intelligence. Vol. 30.*, 2016.

H. van Hasselt, Y. Doron, F. Strub, M. Hessel, N. Sonnerat, and J. Modayil. Deep reinforcement learning and the deadly triad, 2018.

H. van Seijen and R. Sutton. True online td(λ). In *International Conference on Machine Learning*, pages 692–700, 2014.

H. van Seijen and R. Sutton. A deeper look at planning as learning from replay. In *International conference on machine learning*, pages 2314–2322, 2015.

V. N. Vapnik and A. Y. Chervonenkis. On the uniform convergence of relative frequencies of events to their probabilities. In *Measures of complexity*, pages 11–30. Springer, 1971.

A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 2017.

N. Vieillard, B. Scherrer, O. Pietquin, and M. Geist. Momentum in reinforcement learning. In *International Conference on Artificial Intelligence and Statistics*, pages 2529–2538, 2020.

S. Vijayakumar and S. Schaal. Locally weighted projection regression: An o(n) algorithm for incremental real time learning in high dimensional space. *Proceedings of the Seventeenth International Conference on Machine Learning (ICML 2000)*, Vol. 1, 05 2000.

P. Vincent, H. Larochelle, Y. Bengio, and P.-A. Manzagol. Extracting and composing robust features with denoising autoencoders. In *Proceedings of the 25th international conference on Machine learning*, pages 1096–1103, 2008.

O. Vinyals, I. Babuschkin, W. M. Czarnecki, M. Mathieu, A. Dudzik, J. Chung, D. H. Choi, R. Powell, T. Ewalds, P. Georgiev, et al. Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature*, 575(7782):350–354, 2019.

B. Wang, Q. Meng, H. Zhang, R. Sun, W. Chen, and Z.-M. Ma. Momentum doesn't change the implicit bias, 2021.

L. Wasserman. *All of statistics: a concise course in statistical inference.* Springer Science & Business Media, 2013.

C. J. Watkins and P. Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

D. Weininger. Smiles, a chemical language and information system. 1. introduction to methodology and encoding rules. *Journal of chemical information and computer sciences*, 28(1):31–36, 1988.

S. Whiteson, B. Tanner, M. E. Taylor, and P. Stone. Protecting against evaluation overfitting in empirical reinforcement learning. In *2011 IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning (ADPRL)*, pages 120–127. IEEE, 2011.

C. K. Williams and C. Rasmussen. Gaussian processes for regression. In *Neural Information Processing Systems (NeurIPS)*, 1995.

R. J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8(3-4):229–256, 1992a. ISSN 0885-6125. doi: 10.1007/BF00992696.

R. J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992b.

R. J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8(3):229–256, May 1992c. ISSN 1573-0565.

D. R. Wilson and T. R. Martinez. The general inefficiency of batch training for gradient descent learning. *Neural networks*, 16(10):1429–1451, 2003.

S. Witty, J. K. Lee, E. Tosch, A. Atrey, M. Littman, and D. Jensen. Measuring and characterizing generalization in deep reinforcement learning. *arXiv preprint arXiv:1812.02868*, 2018.

Y. Xie, C. Shi, H. Zhou, Y. Yang, W. Zhang, Y. Yu, and L. Li. {MARS}: Markov molecular sampling for multi-objective drug discovery. In *International Conference on Learning Representations*, 2021.

S. Yasini, M. Naghibi-Sistani, and A. Karimpour. Agent-based simulation for blood glucose control in diabetic patients. *International Journal of Applied Science, Engineering and Technology*, 5(1):40–49, 2009.

A. Zhang, N. Ballas, and J. Pineau. A dissection of overfitting and generalization in continuous reinforcement learning. *arXiv preprint arXiv:1806.07937*, 2018a.

A. Zhang, Y. Wu, and J. Pineau. Natural environment benchmarks for reinforcement learning. *CoRR*, 2018b.

C. Zhang, S. Bengio, M. Hardt, B. Recht, and O. Vinyals. Understanding deep learning requires rethinking generalization. *International Conference on Learning Representations*, 2017.

C. Zhang, O. Vinyals, R. Munos, and S. Bengio. A study on overfitting in deep reinforcement learning. *arXiv preprint arXiv:1804.06893*, 2018c.

P. Zhang, X. Li, X. Hu, J. Yang, L. Zhang, L. Wang, Y. Choi, and J. Gao. Vinvl: Revisiting visual representations in vision-language models. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 5579–5588, 2021.

S. Zhang, W. Boehmer, and S. Whiteson. Deep residual reinforcement learning. *19th International Conference on Autonomous Agents and Multiagent Systems, AAMAS 2020. International Foundation for Autonomous Agents and Multiagent Systems (IFAAMAS), 2020.*, 2019.

M. Zhao, Z. Liu, S. Luan, S. Zhang, D. Precup, and Y. Bengio. A consciousness-inspired planning agent for model-based reinforcement learning. *Advances in neural information processing systems*, 2021.