Copyright
by
Abhinav Verma
2021

The Dissertation Committee for Abhinav Verma
certifies that this is the approved version of the following dissertation:

Programmatic Reinforcement Learning

Committee:

Swarat Chaudhuri, Supervisor

Rajeev Alur

Joydeep Biswas

Peter Stone

Programmatic Reinforcement Learning

by

Abhinav Verma

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

August 2021

To my parents, Sangeeta and Lov.

Acknowledgments

गुरू गोविन्द दोऊ खड़े, काके लागूं पांय।
बलिहारी गुरू आपने, गोविन्द दियो बताय।।
-Kabir (15th Century Poet)

I have been extremely fortunate to experience the truth of Kabir's words. Foremost for this, I thank my advisor Swarat Chaudhuri. I consider myself extremely fortunate to be his student. Through the long and winding road of doctoral research, Swarat has been a Friend, Philosopher, and Guide. I hope to emulate his teaching and advising style, and will look up to him as a role model throughout my life.

Many gifted scientists have had a profound impact on my research and career and I am grateful to have been guided by them. I want to thank the members of my thesis committee, Rajeev Alur, Joydeep Biswas, and Peter Stone. Their insightful questions and suggestions have helped me improve this thesis, and will likely inform my research for many years. Yisong Yue's vision and depth of knowledge are truly inspiring, and I am thankful for the opportunity to collaborate with him. Through courses and lab meetings, I have been fortunate to have had access to Moshe Vardi's wisdom on both technical and humanitarian issues. I am grateful to Armando Solar-Lezama for the opportunity to participate in the pioneering Neurosymbolic NSF Expedition.

My education in technical communication has benefited immensely from the expertise of Tina Peterson, and I am very thankful for her guidance in this essential skill.

I am extremely fortunate to have had the opportunity to work with Richard G. Baraniuk, Joel W. Burdick, Isil Dillig, Pushmeet Kohli, Vijayaraghavan Murali, Gabor Orosz, Ankit B. Patel, and Rishabh Singh. Their impact on my research goes well beyond the individual projects that we worked on. Some of my happiest memories from graduate school are the times I collaborated with fellow graduate students and I want to thank Greg Anderson, Richard Cheng, Joshua J. Michalenko, Ameesh Shah, Jennifer J Sun, and Eric Zhan for those opportunities.

During my Ph.D. I had the opportunity to work on some fascinating topics adjacent to my doctoral research while on internships at Microsoft Research and SRI International. I want to thank Marc Brockschmidt, Susmit Jha, and Christoph M. Wintersteiger, for being generous mentors. Susmit's guidance on career choices and research directions have been truly invaluable. I am also grateful for the guidance and support of Christopher Jermaine, Luay K. Nakhleh, and T. S. Eugene Ng.

Moving to Computer Science from a pure math background would not have been possible without the encouragement of many amazing people, and I particularly want to thank Boris Botvinnik, Hsien-Ching Kao, and Sankaran Viswanath for their guidance and support.

The support of friends has been pivotal in shaping my research and career trajectories. I am thankful to Suguman Bansal, Oliver Chang, Dipak Chaudhari, Myra Cheng, Arkabandhu Chowdhury, Jonathan Fernandes, Josh Hoffman, Dimitrije Jankov, Jacqui Li, Yanxin Lu, Kuldeep Meel, Anders Miltner, Rohan Mukherjee, Aditya Shrotri, Sourav Sikdar, Calvin Smith, Yuxin Tang, Yue Wang, and Chenxi Yang for stimulating discussions on research and life.

International students often have to navigate an extra layer of bureaucracy. I am grateful to the amazing staff members whose patience and support made this task manageable. I am particularly indebted to Katie Traughber, Beth Rivera, Lena Sifuentes, and Sherry Nassar for their help.

In my final year I was generously supported by a J.P. Morgan AI Research PhD Fellowship. I am particularly grateful to Manuela M. Veloso and Tucker Balch for this support and for the insightful discussions during my (virtual) visit to J.P. Morgan.

I would certainly not have been able to complete my doctoral studies without the unconditional love and support of my family, my parents Sangeeta Verma and Lov Verma, my brother and sister-in-law Shwetank Verma and Ananya Chandra, and my wife Amita Malik.

Lastly, I would like to express my immense gratitude for the kindness of strangers. The generosity of people I will never know has been invaluable and inspiring. Thank you!

Programmatic Reinforcement Learning

Publication No. ____________________

Abhinav Verma, Ph.D.
The University of Texas at Austin, 2021

Supervisor: Swarat Chaudhuri

Programmatic Reinforcement Learning is the study of learning algorithms that can leverage partial symbolic knowledge provided in expressive high-level domain specific languages. The aim of such algorithms is to learn agents that are reliable, secure, and transparent. This means that such agents can be expected to learn desirable behaviors with limited data, while provably maintaining some essential correctness invariant, and providing insights into their decision mechanisms which can be understood by humans. Contrasted with the popular Deep Reinforcement Learning paradigm, where the learnt policy is represented by a neural network, programmatic representations are more easily interpreted and more amenable to verification by scalable symbolic methods. The interpretability and verifiability of these policies provides the opportunity to deploy reinforcement learning based solutions in safety critical environments. In this dissertation, we formalize the concept of Programmatic Reinforcement Learning, and introduce algorithms that integrate

policy learning with principled mechanisms that incorporate domain knowledge. An analysis of the presented algorithms demonstrates that they posses robust theoretical guarantees and are capable of impressive performance in challenging reinforcement learning environments.

Table of Contents

Acknowledgments v

Abstract viii

List of Tables xiii

List of Figures xv

Chapter 1. Introduction 1

1.1 Reinforcement Learning 4

1.2 Deep Reinforcement Learning 6

1.3 Programmatic Reinforcement Learning 8

Chapter 2. Background 12

2.1 Interpretable Machine Learning. 12

2.2 Verifiable Machine Learning. 13

2.3 Reinforcement Learning Theory 14

Chapter 3. Programmatic Policy Learning 16

3.1 Domain Specific Languages 16

3.2 Programmatic Policy Representations 18

3.3 Neurosymbolic Policy Representations 23

Chapter 4. Imitation Projected Programmatic Policies 25

4.1 Programmatic Representations as Constraints 27

4.2 Neurosymbolic Policies via Mirror Descent 30

4.3 Summary and Practical Considerations 33

Chapter 5. Projection via Program Synthesis **37**
5.1 Program Synthesis Overview 38
5.2 Neurally Directed Program Search 40
5.3 Distillation into Decision Trees 44

Chapter 6. Theoretical Analysis **47**
6.1 Motivation 47
6.2 Background and Results 50
6.2.1 Propel as (Approximate) Functional Mirror Descent . 51
6.2.2 Finite-Sample Analysis under Vanilla Policy Gradient Update and DAgger Projection 53
6.2.3 Closing the gap between the gradient estimates 54
6.3 Details and Proofs 57
6.3.1 Expected Regret Bound under Noisy Policy Gradient Estimates and Projection Errors 59
6.3.2 Finite-Sample Analysis 64
6.3.3 Defining a consistent approximation of the gradient . . . 70

Chapter 7. Experiments and Implementation Details **79**
7.1 Environments for Experiments 79
7.2 Experimental Analysis of Ndps 83
7.2.1 Evaluating Performance 83
7.2.2 Qualitative Analysis of the Programmatic Policy 85
7.2.3 Parameter Optimization 91
7.3 Experimental Analysis of Propel 94
7.3.1 Evaluating Performance 94
7.3.2 Variance Reduction 97
7.3.3 Qualitative Analysis 99

Chapter 8. Conclusion and Future Work **103**
8.1 Neurosymbolic Learning 104
8.2 Applications 105
8.3 Theoretical Foundations 107

Bibliography **109**

Vita **126**

List of Tables

7.1 Performance results in Torcs. Lap time is given in Minutes:Seconds. Timeout indicates that the synthesizer did not return a program that completed the race within the specified timeout. 85

7.2 Smoothness measure of agents in Torcs, given by the standard deviation of the steering actions during a complete race. Lower values indicate smoother steering. 86

7.3 Partial observability results in Torcs after blocking sensors {RPM, TrackPos} . For each track and block probability we give the distance, in meters, raced by the program before crashing. 88

7.4 Transfer results with training on CG-Speedway-1. 'Cr' indicates that the agent crashed after racing the specified distance. . . . 89

7.5 Transfer results with training on Aalborg. 'Cr' denotes the agent crashed, after racing the specified distance. 89

7.6 Performance results in Torcs over 25 random seeds. Each entry reports median lap time in seconds over all the seeds (lower is better). A lap time of Cr indicates the agent crashed and could not complete a lap for more than half the seeds. 96

7.7 Performance results in Torcs over 25 random seeds. Each entry reports the ratio of seeds that result in crashes (lower is better). 97

7.8 Generalization results in Torcs, where rows are training and columns are testing tracks. Each entry is formatted as Propel-Prog / DDPG, and the number reported is the median lap time in seconds over all the seeds (lower is better). Cr indicates the agent crashed and could not complete a lap for more than half the seeds. 100

7.9 Generalization results in Torcs for PropelTree, where rows are training and columns are testing tracks. The number reported is the median lap time in seconds over all the seeds (lower is better). Cr indicates the agent crashed and could not complete a lap for more than half the seeds. 100

7.10 Performance results in Torcs of Propel agents initialized with neural policies obtained via Ddpg, over 25 random seeds. Each entry reports the median lap time in seconds over all the seeds (lower is better). A lap time of Cr indicates the agent crashed and could not complete a lap for more than half the seeds. . . 102

7.11 Performance results in TORCS of PROPEL agents initialized with neural policies obtained via DDPG, over 25 random seeds. Each entry reports the ratio of seeds that result in crashes (lower is better). 102

List of Figures

1.1 Depiction of the RL interaction loop. 5
1.2 Policy learning in a Markov environment. 6
1.3 A DNN used to learn a policy in a RL environment. 7
1.4 An interpretable program for acceleration, automatically discovered in our framework. 8

3.1 Programmatic policy representation for the reinforcement learning problem. 17
3.2 Syntax of the policy language. Here, E and α represent expressions that evaluate to atoms and histories. 20
3.3 A programmatic policy for acceleration, automatically discovered by the NDPS algorithm. 23
3.4 Depiction of Neurosymbolic policy learning. 24

4.1 Depicting the PROPEL framework. 26
4.2 A high-level syntax for programmatic policies, inspired by [95]. A policy $\pi(s)$ takes a state s as input and produces an action a as output. b represents boolean expressions; ϕ is a boolean-valued operator on states; Op is an operator that combines multiple policies into one policy; BOp is a standard boolean operator; and $\oplus_\theta$ is a "library function" parameterized by θ. 29
4.3 A programmatic policy for acceleration in TORCS [96], automatically discovered by PROPEL. s[`TrackPos`] represents the most recent reading from sensor `TrackPos`. 29
4.4 Neurosymbolic policy learning via mirror descent in the PROPEL framework. 31
4.5 The main components of the PROPEL framework and their interactions. 34

5.1 Schematic of programmatic policy learning via the NDPS algorithm. 40

7.1 Screenshot of a car racing in TORCS. The learning agent has access to the environment state through sensors that provide information about the car and track. 80

7.2 Slice of steering actions taken by the DRL and NDPS agents, during the CG-Speedway-1 race. This figure demonstrates that the NDPS agent drives more smoothly. 87

7.3 Distance raced by the agents as the block probability increases for a particular sensor(s) on Aalborg. The NDPS agent is more robust to blocked sensors. 88

7.4 A programmatic policy for steering, automatically discovered by the NDPS algorithm with training on Aalborg. 93

7.5 A programmatic policy for acceleration, automatically discovered by the NDPS algorithm with training on CG-Speedway-1. . . . 93

7.6 A programmatic policy for steering, automatically discovered by the NDPS algorithm with training on CG-Speedway-1. 93

7.7 Median lap-time improvements during multiple iterations of PropelProg over 25 random seeds. 95

7.8 Median number of crashes during training of DDPG and PropelProg over 25 random seeds. 95

7.9 Reward variance in TORCS over 25 random seeds for DDPG, Propel, and TPSR. The variance is calculated over the performance of all the learning agents for a particular algorithm (each initialized with a different random seed) after each training episode. 98

7.10 Learning results for neurosymbolic policies. (a) Reward improvement over fixed programmatic policy with different set values for λ or an adaptive λ. The right plot is a zoomed-in version of the left plot without variance bars for clarity. (b) Performance and variance in the reward as a function of the regularization λ, across different runs of the algorithm using random initializations/seeds. Dashed lines show the performance (i.e. reward) and variance using the adaptive weighting strategy. 99

Chapter 1

Introduction

Recent advances in machine learning have created the possibility of a Software 2.0 revolution, wherein code is generated based on the optimization of an evaluation criterion. However, a key requirement to real-world deployments of such software is generating learnt models that can be trusted by society. Current machine learning techniques rely heavily on Deep Neural Networks based models, which have significant fundamental drawbacks that make reliable learning difficult and learnt models susceptible to catastrophic failures. In some real world deployments of such models, bad outcomes have led to death and disability, thus eroding the public's trust in Artificial Intelligence (AI). The goal of my research is to generate trustworthy AI models, by integrating partial domain knowledge and experience based neural learning.

One avenue for progress is to combine ideas from formal methods and machine learning to efficiently build models that are reliable, transparent, and secure. This means that such systems can be expected to learn desirable behaviors with limited data, while provably maintaining some essential correctness invariant and generating models whose decisions can be understood by humans. I believe that we can achieve these goals via Neurosymbolic learning, which

establishes connections between the symbolic reasoning and inductive learning paradigms of artificial intelligence.

Current machine learning models are dominated by Deep Neural Networks (DNN), because they are capable of leveraging gradient-based algorithms to optimize a specific objective. However, neural models are considered "black-boxes" and are often considered untrustworthy due to the following drawbacks:

1. Hard to interpret: this makes these models hard to audit and debug.
2. Hard to formally verify: due to the lack of abstractions in neural models they are often too large to verify for desirable behavior using automated reasoning tools.
3. Unreliable: neural models have notoriously high levels of variability, to the extent that the random initialization of the weights can determine whether the learner finds a useful model.
4. Lack of domain awareness: neural models lack the ability to bias the learner with commonsense knowledge about the task or environment.

While these issues are well acknowledged in the ML community, most existing approaches tackle these problems individually and are unsuited for creating models that address all four drawbacks simultaneously. For example, existing interpretability tools do not provide a mechanism to make the network more amenable to formal verification. DNN verification techniques suffer scalability issues that reduce their applicability. Known regularization techniques

often introduce a bias whose effects are hard to interpret or verify. And finally, domain awareness is sometimes implicitly encoded by pre-training on related tasks, but this pre-training is computationally expensive and has relatively few theoretical guarantees.

One promising avenue is to address these four drawbacks simultaneously by automatically generating Neurosymbolic Models. These models combine learnt neural models with partial symbolic knowledge expressed via programs in a Domain Specific Language (DSL). At a high level, the neural model can perform learning via gradient-based methods and this information is then distilled into a constrained programmatic model. The constraints act as a mechanism to introduce symbolic domain knowledge into the learning process. The two models can be combined in a variety of ways, which provide a technique to balance the relative benefits and drawbacks of each. In this thesis, we present work that establishes that neurosymbolic models can provide a principled mechanism to combat all four of the above shortcomings of DNN based models in the reinforcement learning paradigm.

The intuition behind this work is that structured programs in a high level DSL have four key benefits. First, the DSL can be designed to be human-readable and is hence more interpretable than a DNN. Second, due to the availability of higher-level abstractions these models have parsimonious representations and are hence more amenable to formal verification techniques which can reason about the learned models and check consistency with desirable properties. Third, the DSL can be used to provide primitives that act as

regularizers during learning, hence creating a more reliable model. Finally, the language can be used to encode commonsense knowledge thus providing a mechanism to programmatically modify the learner's inductive bias.

Concretely, we have developed these ideas in the setting of reinforcement learning where we learned agent policies in high-level interpretable Domain Specific Languages. These programmatic policies are easy to formally verify and clear some significant performance bars, including out-performing the state-of-the-art RL methods on a challenging car racing simulator. A thorough theoretical analysis, of the neurosymbolic learning approach used to generate these policies, shows that this approach is generally applicable to various RL environments.

1.1 Reinforcement Learning

Artificial Intelligence (AI) can be broadly understood as the study of algorithms that help machines mimic human behavior. Within AI, the sub-field of Machine Learning (ML) is where machines learn without explicit instructions, in other words they learn from some form of data without explicit instructions that dictate their behavior. Reinforcement Learning is the branch of Machine Learning where we want machines to learn from interactions with the world.

Specifically, in RL we try and communicate our intent in the form of rewards from the environment and want the machine to learn how to achieve these rewards. The learner or agent, observes the state of the environment, takes an action, then observes how the action changed the state of the environment,

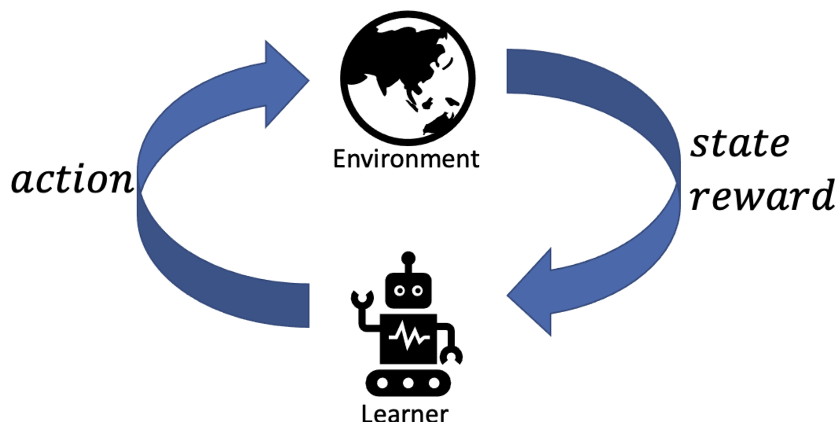

Figure 1.1: Depiction of the RL interaction loop.

and occasionally receives a reward from the environment. This interaction is depicted in Figure 1.1.

At a very high level, the aim of the entire field of reinforcement learning is to create learning algorithms that generate optimal policies for RL environments, as depicted in Figure 1.2. A policy is simply a function that maps states to actions, and it is considered optimal if it maximizes some notion of an accumulated reward. A popular choice for maximization is the expected discounted aggregate reward, as shown in Figure 1.2.

Many techniques have been proposed for a wide variety of environments, and reinforcement learning in general remains an active area of research. In this thesis when discussing existing algorithms we will primarily focus on policy gradient methods. In a nutshell, policy gradient methods optimize the policy directly, by considering a parametric representation of the policy and

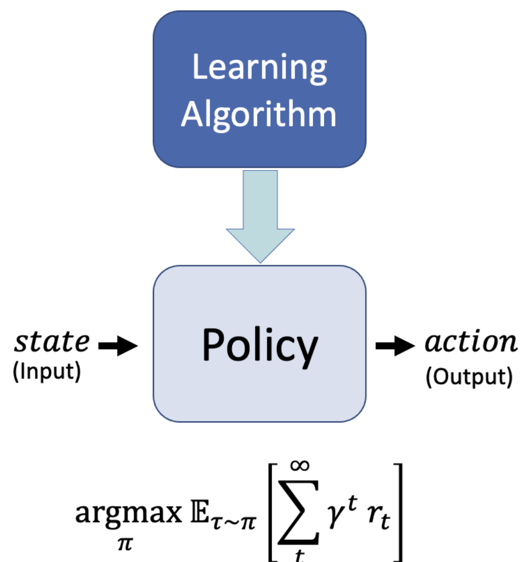

Figure 1.2: Policy learning in a Markov environment.

then performing gradient ascent with respect to the optimization objective. The calculation of the gradient is made tractable by the Policy Gradient Theorem [90]. This theorem simplifies the gradient computation by removing its dependence on the state distribution induced by the policy.

1.2 Deep Reinforcement Learning

Many recent advances in reinforcement learning have been through techniques that rely on a Deep Neural Networks (DNN). The key common idea in many of these techniques is to represent the policy function via a DNN, and then train it end-to-end to optimize the reward objective, as depicted in Figure 1.3. This has been a fruitful choice because neural networks are

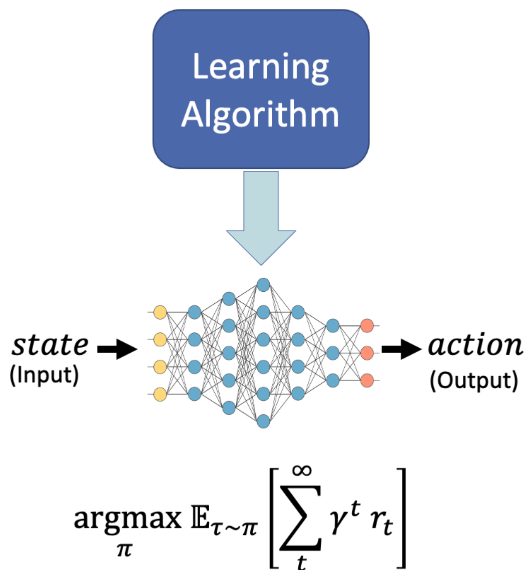

Figure 1.3: A DNN used to learn a policy in a RL environment.

universal function approximators, and they are capable of leveraging gradient-based algorithms to perform optimization efficiently. Recent advances in GPU accelerated hardware, which can be used to efficiently train large neural networks, have further improved the performance and applicability of such approaches.

Deep neural network based policy gradient techniques have been used very successfully for many applications, and particularly for continuous control in simulations. These techniques use many innovative mechanisms to incorporate neural networks as approximators for various functions in the policy search process. In this thesis we will not be delving into the details of these algorithms, and will only use them as experimental baselines, or as modules in

larger frameworks.

1.3 Programmatic Reinforcement Learning

The Programmatic Reinforcement Learning (PRL) framework aims to simultaneously tackle the four primary drawbacks of black-box policies. In summary, we place syntactic restrictions on the policy via a user specified DSL. Our goal is to automatically find a program in this language, which maximizes the agent's expected aggregate reward in the environment. An example of this approach, is to synthesize programs that control a car's acceleration and steering to drive it around a track. Figure 1.4 shows the kind of high-level program our method finds for acceleration, when the DSL provides Propotional-Integral-Derivative (PID) controllers as primitives in the language. In general, the DSL is designed to provide high-level abstractions that are known to be useful for the underlying domain. The automatically generated programs are hence parsimoniously represented in a structured programming language, which is similar to how a human expert would write such code.

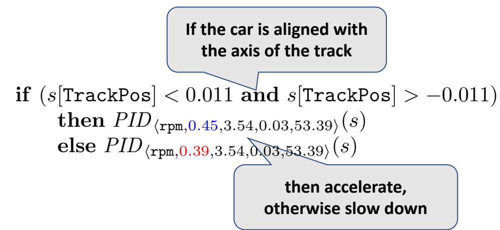

Figure 1.4: An interpretable program for acceleration, automatically discovered in our framework.

A key challenge in PRL is that the space of programs is typically vast and non-smooth making direct search intractable. Our approach to this question,

an algorithm called Neurally Directed Program Search (NDPS), uses deep-RL to compute an initial approximation of the desired program, then uses this neural net as an "oracle" that guides program synthesis. This technique allows us to leverage recent advancements in quantitative program synthesis and SMT solvers.

Neurosymbolic Learning

As the NDPS algorithm requires a neural oracle, it can find performant policies only when deep-RL techniques are able to find performant policies independently. To address this shortcoming, we developed a novel meta-algorithm called Imitation-Projected Programmatic Reinforcement Learning (PROPEL), which is based on mirror descent, program synthesis, and imitation learning. The PROPEL framework leverages neurosymbolic learning to generate programmatic policies, by creating a neurosymbolic policy class which mixes neural and programmatic policy representations. This allows us to cast our learning task as optimization in a constrained policy space, and solve this problem using a "update-and-project" perspective that takes a gradient step into the unconstrained neurosymbolic space and then projects back onto the constrained programmatic space. Essentially, the PROPEL algorithm establishes a synergistic relationship between deep-RL and program synthesis, using synthesized programs to regularize deep-RL and using the gradients available to deep-RL to improve the quality of synthesized programs. This principled mechanism to create a neurosymbolic learner integrates symbolic knowledge

with gradient-based optimization.

The domain knowledge embodied in the programming language acts as a form of regularization. This allows us to prove that neurosymbolic models can learn more reliably than traditional deep-RL methods. A thorough theoretical analysis of Propel characterizes the impact of approximate gradients and projections, providing promising expected regret bounds and finite-sample guarantees with the minimal assumption that the projection error is bounded. This analysis provides confidence that the Propel framework can be reliably applied to a variety of RL environments.

Thesis Statement:

Programmatic reinforcement learning serves as a principled mechanism to incorporate partial symbolic knowledge into learning algorithms providing reliability, interpretability and verifiability.

The remaining document is organized as follows. Chapter 2 discusses background and related work. Chapter 3 formalizes the programmatic reinforcement learning framework. Chapter 4 presents Propel, a mirror-descent inspired meta-learning algorithm for programmatic reinforcement learning. Chapter 5 discusses using program synthesis to perform the projection step in the Propel framework. Chapter 6 presents a theoretical analysis of the Propel framework and provides insights that can help inform implementation design decisions. Chapter 7, provides a through empirical evaluation of the previously presented algorithms. Finally, in Chapter 8 we summarize the

research presented in this thesis and highlight some potential future directions leading to my long-term research goals beyond this dissertation.

Chapter 2

Background

In this chapter, we review prior work relevant to programmatic reinforcement learning specifically. We also discuss some relevant literature from the broader machine learning and formal methods communities.

2.1 Interpretable Machine Learning.

Many recent efforts in deep learning aim to make deep networks more interpretable (e.g. [66], [62]). There are three key approaches explored for interpreting DNNs: i) generate input prototypes in the input domain that are representatives of the learned concept in the abstract domain of the top-level of a DNN, ii) explaining DNN decisions by relevance propagation and computing corresponding representative concepts in the input domain, and iii) Using symbolic techniques to explain and interpret a DNN. See for example, [38], [98], [81], [56]. Our work differs from these approaches in that we are replacing the DRL model with human readable source code, that is programmatically synthesized to mimic the policy found by the neural network. Working at this level of abstraction provides a method to apply existing synthesis techniques to the problem of making DRL models interpretable.

2.2 Verifiable Machine Learning.

Many recent techniques attack the problem of verifying neural networks directly, like [49, 43, 44, 50]. These have been used to verify several properties of DNN-based systems, like airborne collision avoidance systems, autonomous car controllers, etc. Unlike these techniques, our framework generates interpretable program source code as output, where we can use traditional symbolic program verification techniques [51] to prove program properties.

Constrained Policy Learning. Constrained policy learning has seen increased interest in recent years, largely due to the desire to impose side guarantees such as stability and safety on the policy's behavior. Broadly, there are two approaches to imposing constraints: specifying constraints as an additional cost function [2, 58], and explicitly encoding constraints into the policy class [3, 57, 25, 26, 15]. In some cases, these two approaches can be viewed as dual of each other. For instance, recent work that uses control-theoretic policies as a functional regularizer [57, 25] can be viewed from the perspective of both regularization (additional cost) and an explicitly constrained policy class (a specific mix of neural and control-theoretic policies). We build upon this perspective to develop the gradient update step in our approach. [97, 52]

2.3 Reinforcement Learning Theory

The mirror descent framework has previously used to analyze and design RL algorithms. For example, Thomas et al. [93] and Mahadevan and Liu [64] use composite objective mirror descent, or COMID [32], which allows incorporating adaptive regularizers into gradient updates, thus offering connections to either natural gradient RL [93] or sparsity inducing RL algorithms [64]. Unlike in our work, these prior approaches perform projection into the same native, differentiable representation. Also, the analyses in these papers do not consider errors introduced by hybrid representations and approximate projection operators. However, one can potentially extend our approach with versions of mirror descent, e.g., COMID, that were considered in these efforts.

RL using Imitation Learning. There are two ways to utilize imitation learning subroutines within RL. First, one can leverage limited-access or sub-optimal experts to speed up learning [73, 24, 20, 87, 97, 52]. Second, one can learn over two policy classes (or one policy and one model class) to achieve accelerated learning compared to using only one policy class [67, 23, 88, 22]. Our approach has some stylistic similarities to previous efforts [67, 88] that use a richer policy space to search for improvements before re-training the primary policy to imitate the richer policy. One key difference is that our primary policy is programmatic and potentially non-differentiable. A second key difference is that our theoretical framework takes a functional gradient descent perspective — it would be interesting to carefully compare with previous analysis techniques

to find a unifying framework.

Chapter 3

Programmatic Policy Learning

Deep reinforcement learning (DRL) has had a massive impact on the field of machine learning and has led to remarkable successes in the solution of many challenging tasks [65, 83, 84]. While neural networks have been shown to be very effective in learning good policies, the expressivity of these models makes them difficult to interpret or to be checked for consistency for some desired properties, and casts a cloud over the use of such representations in safety-critical applications.

3.1 Domain Specific Languages

Motivated to overcome this problem, we propose a learning framework, called *Programmatically Reinforcement Learning* (PRL), that is based on the idea of learning policies that are represented in a human-readable language. The PRL framework is parameterized on a high-level *Domain Specific Language* (DSL) for policies. A problem instance in PRL is similar to a one in traditional RL, but also includes a syntactically defined set of programmatic policies in this language. The objective is to find a program in this set with maximal long-term reward.

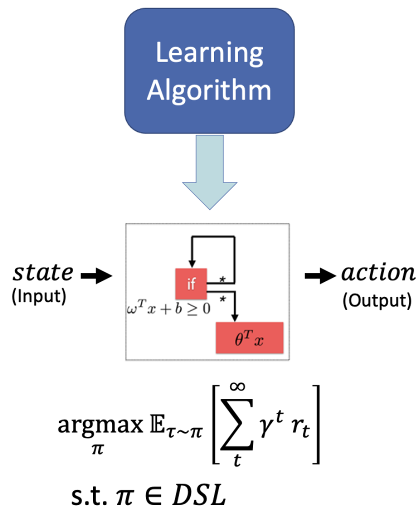

Figure 3.1: Programmatic policy representation for the reinforcement learning problem.

Intuitively, the policy programming language characterizes what we consider "interpretable". In addition to interpretability, the language allows for three key additional benefits. First, the language can be used to implicitly encode the learner's inductive bias that will be used for generalization. Second, the language can allow effective pruning of undesired policies to make the search for a good policy more efficient. Finally, learning a policy program in the language allows us to use symbolic program verification techniques to reason about the learnt policies and check consistency with certain desirable properties.

3.2 Programmatic Policy Representations

We model a reinforcement learning setting as a *Partially Observable Markov Decision Process (*POMDP*)* $M = (\mathcal{S}, \mathcal{A}, \mathcal{O}, T(\cdot|s,a), Z(\cdot|s), r, in, \gamma)$. Here, $\mathcal{S}$ is the set of (environment) states. $\mathcal{A}$ is the set of *actions* that the learning agent can perform, and $\mathcal{O}$ is the set of *observations* about the current state that the agent can make. An agent action a at the state s causes the environment state to change probabilistically, and the destination state follows the distribution $T(\cdot|s,a)$. The probability that the agent makes an observation o at state s is $Z(o|s)$. The *reward* that the agent receives on performing action a in state s is given by $r(s,a)$. in is the initial distribution over environment states. Finally, $0 < \gamma < 1$ is a real constant that is used to define the agent's aggregate reward over time.

A *history* of M is a sequence $h = o_0, a_0, \ldots, a_{k-1}, o_k$, where o_i and a_i are, respectively, the agent's observation and action at the i-th time step. Let $\mathcal{H}_M$ be the set of histories in M. A *policy* is a function $\pi : \mathcal{H}_M \to \mathcal{A}$ that maps each history as above to an action a_k. For each policy, we can define a set of histories that are possible when the agent follows π. We assume a mechanism to simulate the POMDP and sample histories that are possible under a policy. The policy also induces a distribution over possible values of the reward R_i that the agent receives at the i-th time step. The agent's *expected aggregate reward* under π is given by $R(\pi) = \mathbf{E}[\sum_{i=0}^{\infty} \gamma^i R_i]$. The goal in reinforcement learning is to discover a policy π^* that maximizes $R(\pi)$.

A Programming Language for Policies. The distinctive feature of PRL is that policies here are expressed in an interpretable programming language. Such a language can be defined in many ways. However, to facilitate search through the space of programs expressible in the language, it is desirable for the language to express computations as compactly and canonically as possible. Because of this, we propose to express parameterized policies using a functional programming language based on a small number of side-effect-free combinators. It is known from prior work on program synthesis [34] that such languages offer natural advantages in program synthesis.

We collectively refer to observations and actions, as well as auxiliary integers and reals generated during computation, as *atoms*. Our language considers two kinds of data: atoms and sequences of atoms (including histories). We assume a finite set of *basic operators* over atoms that is rich enough to capture all common operations on observations and actions.

Figure 3.2 shows the syntax of this language. The nonterminals E and α represent expressions that evaluate to atoms and histories, respectively. We sketch the semantics of the various language constructs below.

- c ranges over a universe of numerical constants, and Op is a basic operator
- $[\,]$ is the empty sequence, **hd** returns the element in an input sequence representing the most recent time point, and **tl** returns the prefix of the sequence up to (and excluding) this element. "**push** e a" evaluates the

$$
\begin{aligned}
E \quad &::= \quad c \mid x \mid Op(E_1, \ldots, E_k) \mid \mathbf{hd}\ \alpha \mid \\
&\qquad \mathbf{fold}\ (\lambda x_1, x_2.\ E_1)\ \alpha \\
\alpha \quad &::= \quad x \mid [\,] \mid \mathbf{tl}\ \alpha_1 \mid \mathbf{push}\ E\ \alpha_1 \mid \mathbf{map}\ (\lambda x.\ E)\ \alpha_1 \mid \\
&\qquad \mathbf{filter}\ (\lambda x.\ E)\ \alpha_1
\end{aligned}
$$

Figure 3.2: Syntax of the policy language. Here, E and α represent expressions that evaluate to atoms and histories.

atom-valued expression e, then puts the result on top of the history to which a evaluates;

- **map**, **fold**, **filter** are the standard higher-order combinators over sequences with the semantics:
  $\mathbf{map}(f, [e_1, \ldots, e_k]) = [f(e_1), \ldots, f(e_k)]$
  $\mathbf{fold}(f, [e_1, \ldots, e_k], e) = f(e_k, f(e_{k-1}, ...f(e_1, e)))$
  $\mathbf{filter}(f, [e_1, \ldots, e_k]) = [e_{f_1}, \ldots, e_{f_j}]$ where for all $1 \leq i \leq j$, $f(e_{f_i})$ is true;
- x, x_1, x_2 are variables. As usual, unbound variables are assumed to be inputs.

The language comes with a type system that distinguishes between different types of atoms, and ensures that language constructs are used consistently. The type system can catch common errors, such as applying **hd** to the empty sequence. This type system identifies a set of expressions whose inputs are histories and outputs are actions. These expressions are known as *programmatic policies*, or simply *programs*.

Sketches. Discovering an optimal programmatic policy from the vast space of legitimate programs is typically impractical without some prior on the shape of target policies. PRL allows the specification of such priors through instance-specific syntactic models called *sketches*.

We define a sketch as a grammar of expressions over atoms and sequences of atoms, obtained by restricting the grammar in Figure 3.2. Just as we defined the set of programmatic policies permitted in PRL, we can define the set of programs permitted by a sketch $\mathcal{S}$. We denote this set by $[\![\mathcal{S}]\!]$.

PRL. The PRL problem can now be stated as follows. Suppose we are given a POMDP M and a sketch $\mathcal{S}$. Our goal is to find a program $e^* \in [\![\mathcal{S}]\!]$ with optimal reward:

$$e^* \quad = \quad \operatorname*{argmax}_{e \in [\![\mathcal{S}]\!]} R(e). \tag{3.1}$$

Example. Now we consider a concrete example of PRL, suppose our goal is to make a (simulated) car complete laps on a track. We want to do so by learning policies for tasks like steering and acceleration. Suppose we know that we could get well-behaved policies by choosing between a set of Proportional Integral Derivative (PID) controllers. However, we do not know the gains for these controllers, and neither do we know the conditions under which we switch from one controller to another. We can express this knowledge using

the following sketch:

$$
\begin{aligned}
E &::= C \mid \textbf{if } B \textbf{ then } E_1 \textbf{ else } E_2 \\
C &::= c_1 + c_2 * (\epsilon - \mathbf{hd}(x_i)) + c_3 * \mathbf{fold}(+, x_i) + \\
&\qquad c_4 * (\mathbf{hd}(\mathbf{tl}(x_i)) - \mathbf{hd}(x_i)) \\
B &::= c_0 + c_1 * \mathbf{hd}(x_1) + \cdots + c_k * \mathbf{hd}(x_k) > 0 \mid \\
&\qquad B_1 \textbf{ or } B_2 \mid B_1 \textbf{ and } B_2.
\end{aligned}
$$

Here, E represents programs permitted by the sketch. The program's input is a history h. We assume that this sequence is split into a set of sequences $\{h_1, \ldots, h_k\}$, where h_i is the sequence of observations from the i-th sensor. The sensor's most recent reading is given by $\mathbf{hd}(h_i)$, and its second most recent reading is $\mathbf{hd}(\mathbf{tl}(h_i))$. The operators $+$, $-$, $*$, $>$, and if-then-else are as usual. The program (optionally) evaluates a set of boolean conditions (B) over the current sensor readings, then chooses among a set of PID Controllers (C). In the expression C for PID Controllers, ϵ is a known constant and represents the target for the controller, and the operator **fold** is used to encode the integral term. The symbols c_i are real-valued parameters.

The program in Figure 3.3 shows the body of a policy for acceleration that the NDPS algorithm finds given this sketch. The program has been "decompiled", using standard PL techniques, into a program from a language with higher readability. The program's input consists of histories for 29 sensors; however, only two of them, `TrackPos` and `RPM`, are actually used in the program. While the sensor `TrackPos` (for the position of the car relative to the track

axis) is used to decide which controller to use, only the RPM sensor is needed to calculate the acceleration.

$$
\begin{array}{l}
\textbf{let } P = \textbf{hd}(h_{\texttt{RPM}}),\ I = \textbf{fold}(+, h_{\texttt{RPM}}),\ D = (\textbf{hd}(\textbf{tl}(h_{\texttt{RPM}})) - \textbf{hd}(h_{\texttt{RPM}})) \\
\quad \textbf{in} \\
\textbf{if } (0.001 - \textbf{hd}(h_{\texttt{TrackPos}}) > 0) \textbf{ and } (0.001 + \textbf{hd}(h_{\texttt{TrackPos}}) > 0) \\
\qquad \textbf{then } 1.96 + 4.92 * (0.44 - P) + 0.89 * I + 49.79 * D \\
\qquad \textbf{else } \ 1.78 + 4.92 * (0.40 - P) + 0.89 * I + 49.79 * D
\end{array}
$$

Figure 3.3: A programmatic policy for acceleration, automatically discovered by the NDPS algorithm.

3.3 Neurosymbolic Policy Representations

Finding programmatic policies is a challenging problem because the policy space defined by a reasonably expressive DSL is typically vast and non-smooth. Finding good policies in such spaces typically requires combinatorial optimization techniques that are more computationally expensive than the gradient based techniques used by DRL. The success of deep neural networks based policy learning techniques inspires us to leverage neurosymbolic policy representations to learn better programmatic representations. We discuss such techniques in later chapters, a schematic diagram of neurosymbolic policy representations is depicted in Figure 3.4.

Suppose we have a programmatic policy, $\pi : S \to A$, and we want to combine it with a neural policy, f_θ, defined on the same state and action space.

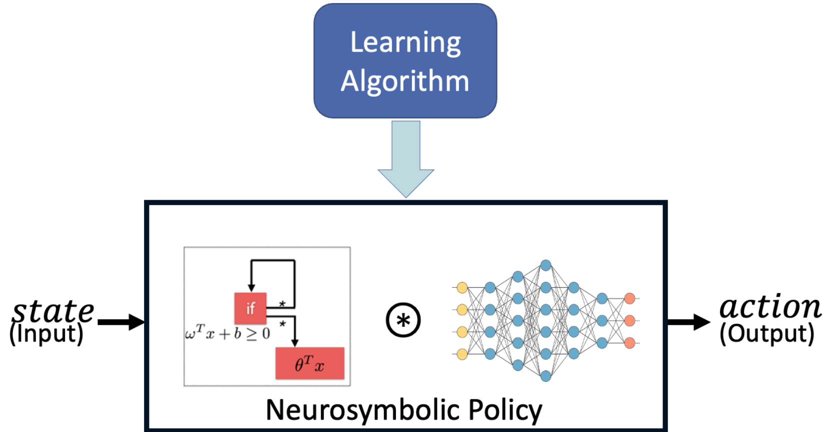

Figure 3.4: Depiction of Neurosymbolic policy learning.

One possible mechanism to combine these policies is:

$$h(s) = \frac{1}{1+\lambda} f_\theta(s) + \frac{\lambda}{1+\lambda} \pi(s) \tag{3.2}$$

where we assume a continuous, convex action space. We refer to $h(s)$ as the *neurosymbolic policy*. For a fixed programmatic policy π, the neurosymbolic policy is equivalent to placing a functional regularizer on the neural policy, f_θ, with regularizer weight λ. Intuitively, λ should be large when the neural controller is highly uncertain, and it should decrease as we become more confident in the neural controller. Details about the effects of this regularization, along with an algorithm to automatically tune λ, and theoretical results on the resulting bias-variance trade-off are presented in [25].

Chapter 4

Imitation Projected Programmatic Policies

A growing body of work [95, 11, 99] investigates reinforcement learning (RL) approaches that represent policies as programs in a symbolic language, e.g., a domain-specific language for composing control modules such as PID controllers [6]. Short programmatic policies offer many advantages over neural policies discovered through deep RL, including greater interpretability, better generalization to unseen environments, and greater amenability to formal verification. These benefits motivate developing effective approaches for learning such programmatic policies.

However, programmatic reinforcement learning (PRL) remains a challenging problem, owing to the highly structured nature of the policy space. Recent state-of-the-art approaches employ program synthesis methods to imitate or distill a pre-trained neural policy into short programs [95, 11]. However, such a distillation process can yield a highly suboptimal programmatic policy — i.e., a large distillation gap — and the issue of direct policy search for programmatic policies also remains open.

In this section, we present PROPEL (Imitation-**Pro**jected **P**rogrammatic R**e**inforcement **L**earning), a new learning meta-algorithm for PRL, as a response

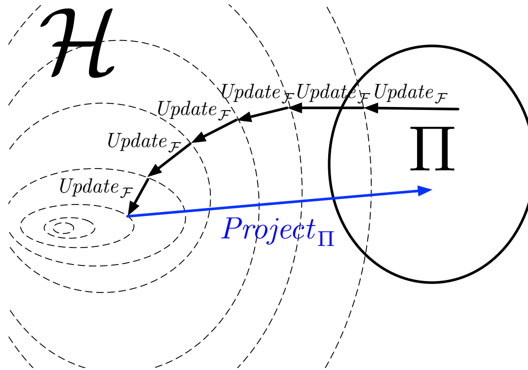

Figure 4.1: Depicting the PROPEL framework.

to this challenge. The design of PROPEL is based on three insights that enables integrating and building upon state-of-the-art approaches for policy gradients and program synthesis. First, we view programmatic policy learning as a constrained policy optimization problem, in which the desired policies are constrained to be those that have a programmatic representation. This insight motivates utilizing constrained mirror descent approaches, which take a gradient step into the unconstrained policy space and then project back onto the constrained space. Second, by allowing the unconstrained policy space to have a mix of neural and programmatic representations, we can employ well-developed deep policy gradient approaches [91, 60, 78, 79, 25] to compute the unconstrained gradient step.

Third, we define the projection operator using program synthesis via imitation learning [95, 11], in order to recover a programmatic policy from the unconstrained policy space. Our contributions can be summarized as:

- We present Propel, a novel meta-algorithm that is based on mirror descent, program synthesis, and imitation learning, for Prl.
- On the theoretical side, we show how to cast Propel as a form of constrained mirror descent. We provide a thorough theoretical analysis characterizing the impact of approximate gradients and projections. Further, we prove results that provide expected regret bounds and finite-sample guarantees under reasonable assumptions.
- On the practical side, we provide a concrete instantiation of Propel and evaluate it in the challenging car-racing domain Torcs [96]. The experiments show significant improvements over state-of-the-art approaches for learning programmatic policies.

4.1 Programmatic Representations as Constraints

The problem of programmatic reinforcement learning (Prl) consists of a Markov Decision Process (Mdp) $\mathcal{M}$ and a programmatic policy class Π. The definition of $\mathcal{M} = (\mathcal{S}, \mathcal{A}, P, c, p_0, \gamma)$ is standard [90], with $\mathcal{S}$ being the state space, $\mathcal{A}$ the action space, $P(s'|s, a)$ the probability density function of transitioning from a state-action pair to a new state, $c(s, a)$ the state-action cost function, $p_0(s)$ a distribution over starting states, and $\gamma \in (0, 1)$ the discount factor. A policy $\pi : \mathcal{S} \to \mathcal{A}$ (stochastically) maps states to actions. We focus on continuous control problems, so $\mathcal{S}$ and $\mathcal{A}$ are assumed to be continuous

spaces. The goal is to find a programmatic policy $\pi^* \in \Pi$ such that:

$$\pi^* = \underset{\pi \in \Pi}{\operatorname{argmin}} J(\pi), \qquad \text{where: } J(\pi) = \mathbf{E}\left[\sum_{i=0}^{\infty} \gamma^i c(s_i, a_i \equiv \pi(s_i))\right], \tag{4.1}$$

with the expectation taken over the initial state distribution $s_0 \sim p_0$, the policy decisions, and the transition dynamics P. One can also use rewards, in which case (4.1) becomes a maximization problem.

Programmatic Policy Class. A programmatic policy class Π consists of policies that can be represented parsimoniously by a (domain-specific) programming language. Recent work [95, 11, 99, 92] indicates that such policies can be easier to interpret and formally verify than neural policies, and can also be more robust to changes in the environment.

In this section, we consider two concrete classes of programmatic policies. The first, a simplification of the class considered in Verma et al. [95], is defined by the modular, high-level language in Figure 4.2. This language assumes a library of parameterized functions $\oplus_\theta$ representing standard controllers, for instance Proportional-Integral-Derivative (PID) [8] or bang-bang controllers [14]. Programs in the language take states s as inputs and produce actions a as output, and can invoke fully instantiated library controllers along with predefined arithmetic, boolean and relational operators. The second, "lower-level" class, from Bastani et al. [11], consists of decision trees that map states to actions.

Example. Consider the problem of learning a programmatic policy, in the language of Figure 4.2, that controls a car's accelerator in the TORCS

$$
\begin{aligned}
\pi(s) \quad ::= \quad & a \\
| \quad & Op(\pi_1(s), \ldots, \pi_k(s)) \\
| \quad & \textbf{if } b \textbf{ then } \pi_1(s) \textbf{ else } \pi_2(s) \\
| \quad & \oplus_\theta(\pi_1(s), \ldots, \pi_k(s)) \\
b \quad ::= \quad & \phi(s) \\
| \quad & BOp(b_1, \ldots, b_k)
\end{aligned}
$$

Figure 4.2: A high-level syntax for programmatic policies, inspired by [95]. A policy $\pi(s)$ takes a state s as input and produces an action a as output. b represents boolean expressions; ϕ is a boolean-valued operator on states; Op is an operator that combines multiple policies into one policy; BOp is a standard boolean operator; and $\oplus_\theta$ is a "library function" parameterized by θ.

$$
\begin{aligned}
&\textbf{if } (s[\texttt{TrackPos}] < 0.011 \textbf{ and } s[\texttt{TrackPos}] > -0.011) \\
&\qquad \textbf{then } \texttt{PID}_{\langle \texttt{RPM},0.45,3.54,0.03,53.39\rangle}(s) \textbf{ else } \texttt{PID}_{\langle \texttt{RPM},0.39,3.54,0.03,53.39\rangle}(s)
\end{aligned}
$$

Figure 4.3: A programmatic policy for acceleration in TORCS [96], automatically discovered by PROPEL. s[`TrackPos`] represents the most recent reading from sensor `TrackPos`.

car-racing environment [96]. Figure 4.3 shows a program in our language for this task. The program invokes PID controllers $\texttt{PID}_{\langle j,\theta_P,\theta_I,\theta_D\rangle}$, where j identifies the sensor (out of 29, in our experiments) that provides inputs to the controller, and θ_P, θ_I, and θ_D are respectively the real-valued coefficients of the proportional, integral, and derivative terms in the controller. We note that the program only uses the sensors `TrackPos` and `RPM`. While `TrackPos` (for the position of the car relative to the track axis) is used to decide which controller to use, only the `RPM` sensor is needed to calculate the acceleration.

Learning Challenges. Learning programmatic policies in the contin-

uous RL setting is challenging, as the best performing methods utilize policy gradient approaches [91, 60, 78, 79, 25], but policy gradients are hard to compute in programmatic representations. In many cases, Π may not even be differentiable. For our approach, we only assume access to program synthesis methods that can select a programmatic policy $\pi \in \Pi$ that minimizes imitation disagreement with demonstrations provided by a teaching oracle. Because imitation learning tends to be easier than general RL in long-horizon tasks [89], the task of imitating a neural policy with a program is, intuitively, significantly simpler than the full programmatic RL problem. This intuition is corroborated by past work on programmatic RL [95], which shows that direct search over programs often fails to meet basic performance objectives.

4.2 Neurosymbolic Policies via Mirror Descent

To develop our approach, we take the viewpoint of (4.1) being a constrained optimization problem, where $\Pi \subset \mathcal{H}$ resides within a larger space of policies $\mathcal{H}$. In particular, we will represent $\mathcal{H} \equiv \Pi \oplus \mathcal{F}$ using a mixing of programmatic policies Π and neural polices $\mathcal{F}$. Any mixed policy $h \equiv \pi + f$ can be invoked as $h(s) = \pi(s) + f(s)$. In general, we assume that $\mathcal{F}$ is a good approximation of Π (i.e., for each $\pi \in \Pi$ there is some $f \in \mathcal{F}$ that approximates it well), which we formalize in Section 6.3.

We can now frame our constrained learning problem as minimizing (4.1) over $\Pi \subset \mathcal{H}$, that alternate between taking a gradient step in the general space $\mathcal{H}$ and projecting back down onto Π. This "lift-and-project" perspec-

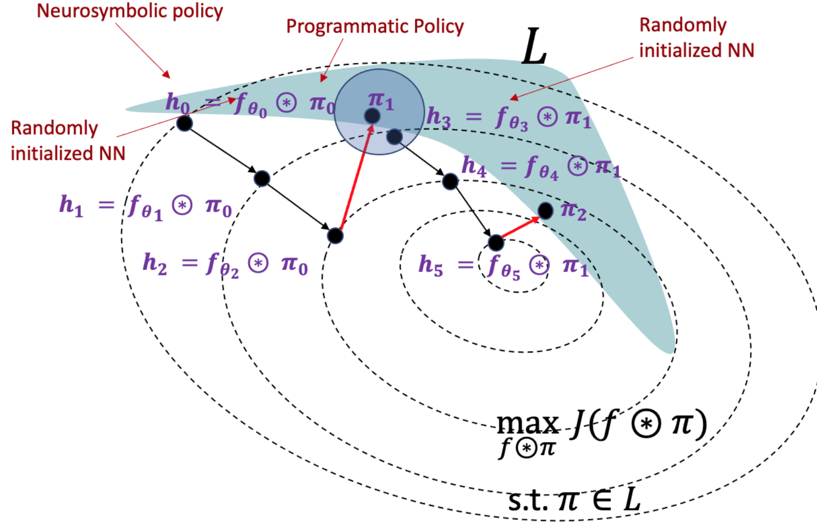

Figure 4.4: Neurosymbolic policy learning via mirror descent in the PROPEL framework.

tive motivates viewing our problem via the lens of mirror descent [69]. In standard mirror descent, the unconstrained gradient step can be written as $h \leftarrow h_{prev} - \eta \nabla_{\mathcal{H}} J(h_{prev})$ for step size η, and the projection can be written as $\pi \leftarrow \text{argmin}_{\pi' \in \Pi} D(\pi', h)$ for divergence measure D.

Our approach, *Imitation-Projected Programmatic Reinforcement Learning* (PROPEL), is outlined in Algorithm 1 (also see Figure 4.1). PROPEL is a meta-algorithm that requires instantiating two subroutines, UPDATE and PROJECT, which correspond to the standard update and projection steps, respectively. PROPEL can be viewed as a form of functional mirror descent with some notable deviations from vanilla mirror descent.

Neurosymbolic policy learning via this framework is depicted in Fig-

Algorithm 1 Imitation-Projected Programmatic Reinforcement Learning (PROPEL)

1: **Input:** Programmatic & Neural Policy Classes: Π & $\mathcal{F}$.
2: **Input:** Either initial π_0 or initial f_0
3: Define joint policy class: $\mathcal{H} \equiv \Pi \oplus \mathcal{F}$ //*$h \equiv \pi + f$ defined as $h(s) = \pi(s) + f(s)$*
4: **if** given initial f_0 **then**
5: $\pi_0 \leftarrow$ PROJECT(f_0) //*program synthesis via imitation learning*
6: **end if**
7: **for** $t = 1, \ldots, T$ **do**
8: $h_t \leftarrow$ UPDATE$_{\mathcal{F}}(\pi_{t-1}, \eta)$ //*policy gradient in neural policy space with learning rate η*
9: $\pi_t \leftarrow$ PROJECT$_{\Pi}(h_t)$ //*program synthesis via imitation learning*
10: **end for**
11: **Return:** Policy π_T

ure 4.4. Here we start with a neurosymbolic policy which consists of a randomly initialized neural component and a programmatic component from the programmatic policy class. We first take gradient updates of the neural component which improves the overall neurosymbolic policy. This improved policy is then projected back to the programmatic class (via program synthesis), to find a more performant programmatic component. These steps can be repeated until convergence, or a fixed computational budget.

UPDATE$_{\mathcal{F}}$. Since policy gradient methods are well-developed for neural policy classes $\mathcal{F}$ (e.g., [60, 78, 79, 41, 31, 25]) and non-existent for programmatic policy classes Π, PROPEL is designed to leverage policy gradients in $\mathcal{F}$ and avoid policy gradients in Π. Algorithm 2 shows one instantiation of UPDATE$_{\mathcal{F}}$. Note that standard mirror descent takes unconstrained gradient steps in $\mathcal{H}$ rather than $\mathcal{F}$, and we discuss this discrepancy between UPDATE$_{\mathcal{H}}$ and UPDATE$_{\mathcal{F}}$ in

Section 6.3.

Algorithm 2 UPDATE$_{\mathcal{F}}$: neural policy gradient for mixed policies

1: **Input:** Neural Policy Class $\mathcal{F}$.
2: **Input:** Reference programmatic policy: π
3: **Input:** Step size: η.
4: **Input:** Regularization parameter: λ
5: Initialize neural policy: f_0 *//any standard randomized initialization*
6: **for** $j = 1, \ldots, m$ **do**
7: $f_j \leftarrow f_{j-1} - \eta\lambda\nabla_{\mathcal{F}} J(\pi + \lambda f_{j-1})$ *//using DDPG [60], TRPO [78], etc., holding π fixed*
8: **end for**
9: **Return:** $h \equiv \pi + \lambda f_m$

PROJECT$_\Pi$. Projecting onto Π can be implemented using program synthesis via imitation learning, i.e., by synthesizing a $\pi \in \Pi$ to best imitate demonstrations provided by a teaching oracle $h \in \mathcal{H}$. Recent work [95, 11, 99] has given practical heuristics for this task for various programmatic policy classes. Algorithm 3 shows one instantiation of PROJECT$_\Pi$ (based on DAgger [75]). One complication that arises is that finite-sample runs of such imitation learning approaches only return approximate solutions and so the projection is not exact. We characterize the impact of approximate projections in Section 6.3.

4.3 Summary and Practical Considerations

The PROPEL approach is at the intersection of three strands of work: (i) program synthesis, (ii) learning policies for sequential decision making, and (iii) constrained learning using approaches such as mirror descent. We

Algorithm 3 PROJECT$_\Pi$: program synthesis via imitation learning

1: **Input:** Programmatic Policy Class: Π.
2: **Input:** Oracle policy: h
3: Roll-out h on environment, get trajectory: $\tau_0 = (s^0, h(s^0), s^1, h(s^1), \ldots)$
4: Create supervised demonstration set: $\Gamma_0 = \{(s, h(s))\}$ from τ_0
5: Derive π_0 from Γ_0 via program synthesis *//e.g., using methods in [95, 11]*
6: **for** $k = 1, \ldots, M$ **do**
7: Roll-out π_{k-1}, creating trajectory: τ_k
8: Collect demonstration data: $\Gamma' = \{(s, h(s)) | s \in \tau_k\}$
9: $\Gamma_k \leftarrow \Gamma' \cup \Gamma_{k-1}$ *//DAgger-style imitation learning [75]*
10: Derive π_k from Γ_k via program synthesis *//e.g., using methods in [95, 11]*
11: **end for**
12: **Return:** π_M

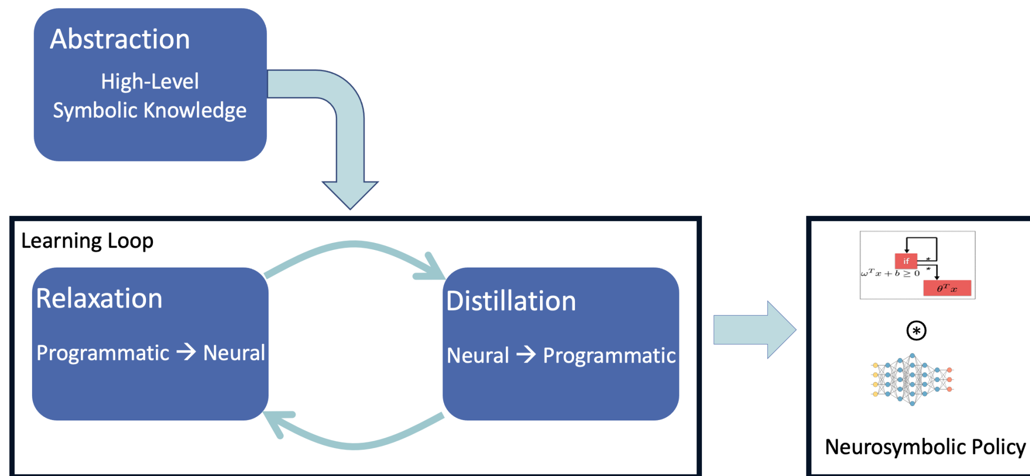

Figure 4.5: The main components of the PROPEL framework and their interactions.

have presented Propel, a meta-algorithm based on mirror descent, program synthesis, and imitation learning, for programmatic reinforcement learning (Prl). The interactions of these components is depicted in Figure 4.5. In Section 6.3 we present theoretical convergence results for Propel, developing novel analyses to characterize approximate projections and biased gradients within the mirror descent framework. In Section 7.3 we validate Propel empirically, and show that it can discover interpretable, verifiable, generalizable, performant policies and significantly outperform the state of the art in Prl.

The central idea of Propel is the use of imitation learning and combinatorial methods in implementing a projection operation for mirror descent, with the goal of optimization in a functional space that lacks gradients. While we have developed Propel in an RL setting, this idea is not restricted to RL or even sequential decision making.

In practice, we often employ multiple gradient steps before taking a projection step (as also described in Algorithm 2), because the step size of individual (stochastic) gradient updates can be quite small. Another issue that arises in virtually all policy gradient approaches is that the gradient estimates can have very high variance [91, 53, 41]. We utilize low-variance policy gradient updates by using the reference π as a proximal regularizer in function space [25].

For the projection step (Algorithm 3), in practice we often retain all previous roll-outs τ from all previous projection steps. It is straightforward to query the current oracle h to provide demonstrations on the states $s \in \tau$ from

previous roll-outs, which can lead to substantial savings in sample complexity with regards to executing roll-outs on the environment, while not harming convergence.

Chapter 5

Projection via Program Synthesis

One of the key technical challenges of enabling PRL is based on the fact that the space of policies permitted in an instance can be vast and nonsmooth, making optimization extremely challenging. To address this, we propose a new algorithm called *Neurally Directed Program Synthesis* (NDPS). The algorithm first uses DRL to compute a neural policy network that has high performance, but may not be expressible in the policy language. This network is then used to direct a local search over programmatic policies. In each iteration of this search, we maintain a set of "interesting" inputs, and update the program so as to minimize the distance between its outputs and the outputs of the neural policy (an "oracle") on these inputs. The set of interesting inputs is updated as the search progresses. This strategy, inspired by imitation learning [75, 77], allows us to perform direct policy search in a highly nonsmooth policy space.

We evaluate our approach in the task of learning to drive a simulated car in the TORCS car-racing environment [96]. Experiments demonstrate that NDPS is able to find interpretable policies that, while not as performant as the policies computed by DRL, pass some significant performance bars. Specifically, in TORCS, our policy syntax allows an unbounded set of programs with branches

guarded by unknown conditions, each branch representing a proportional-integral-derivative (PID) controller [7] with unknown gains. The policy we obtain is able to perform the challenging task of successfully completing a lap of the race, and the use of the neural oracle is key to doing so. Our results also suggest that a well-designed sketch can serve as a regularizer. Due to constraints imposed by the sketch, the policies for TORCS that NDPS learns lead to smoother trajectories than the corresponding neural policies, and can tolerate greater noise. The policies are also more easily transferred to new domains, in particular race tracks not seen during training. Finally, we show, using several properties, that the programmatic policies that we discover are amenable to verification using off-the-shelf symbolic techniques.

5.1 Program Synthesis Overview

Program synthesis is the problem of automatically searching for a program within a language that fits a given specification [40]. Recent approaches to the problem have leveraged symbolic knowledge about program structure [35], satisfiability solvers [86, 45], and meta-learning techniques [68, 70, 28, 10] to generate interesting programs in many domains [4, 71, 5]. In most prior work, the specification is a logical constraint on the input/output behavior of the target program. However, there is also a growing body of work that considers program synthesis modulo optimality objectives [16, 21, 72], often motivated by machine learning tasks [46, 42, 68, 94, 33, 30, 95, 11, 99]. Synthesis of programs that imitates an oracle has been considered in both the logical [45]

and the optimization [95, 11, 99] settings. The projection step in PROPEL builds on this prior work. While our current implementation of this step is entirely symbolic, in principle, the operation can also utilize contemporary techniques for learning policies that guide the synthesis process [68, 10, 82, 59].

Neural Program Synthesis and Induction. Many recent efforts use neural networks for learning programs. These efforts have two flavors. In *neural program induction*, the goal is to learn a network that encodes the program semantics using internal weights. These architectures typically augment neural networks with differentiable computational substrates such as memory (Neural Turing Machines [39]), modules (Neural RAM [55]) or data-structures such as stacks [47], and formulate the program learning problem in an end-to-end differentiable manner. In *neural program synthesis*, the architectures generate programs directly as outputs using multi-task transfer learning (e.g. ROBUSTFILL [29], DEEPCODER [10], BAYOU [68], NEAR [80]), where the network weights are used to guide the program search in a DSL. There have also been some recent approaches to use reinforcement learning for learning to search programs in DSLs [18, 1]. Our approach falls in the category of program synthesis approaches where we synthesize policies in a policy language. However, many of these techniques could also be used in the projection step of the PROPEL framework.

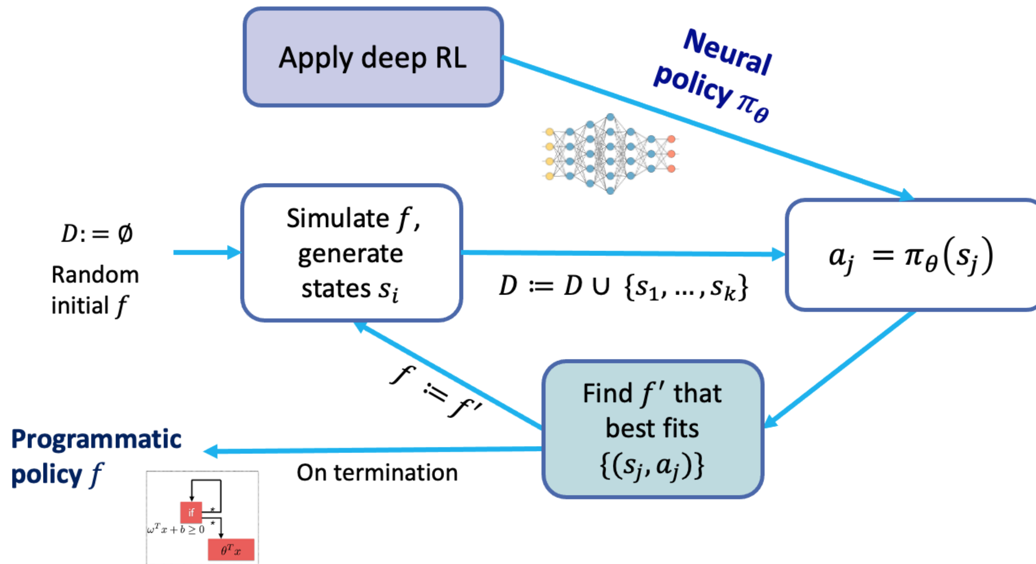

Figure 5.1: Schematic of programmatic policy learning via the NDPS algorithm.

5.2 Neurally Directed Program Search

The NDPS algorithm is a direct policy search that is guided by a neural "oracle". Searching over policies is a standard approach in reinforcement learning. However, the non-smoothness of the space of programmatic policies poses a fundamental challenge to the use of such an approach in PRL. For example, a conceivable way of solving the search problem would be to define a neighborhood relation over programs and perform local search. However, in practice, the objective $R(e)$ of such a search can vary irregularly, leading to poor performance.

In contrast, NDPS starts by using DRL to compute a neural policy

oracle e_{NN} for the given environment. This policy is an approximation of the programmatic policy that we seek to find. To a first approximation, NDPS is a local search over programmatic policies that seeks to find a program e^* that closely imitates the behavior of e_{NN}. The main intuition here is that distance from e_{NN} is a simpler objective than the reward function $R(e)$, which aggregates rewards over a lengthy time horizon. This approach can be seen to be a form of imitation learning [77]. Figure 5.1 shows a schematic representation of this learning algorithm.

The *distance* between e_{NN} and the estimate e of e^* in a search iteration is defined as

$$d(e_{NN}, e) = \sum_{h \in \mathcal{H}} \|e(h) - e_{NN}(h)\|$$

where $\mathcal{H}$ is a set of "interesting" inputs (histories) and $\|\cdot\|$ is a suitable norm. During the iteration, we search the neighborhood of e for a program e' that minimizes this distance. At the end of the iteration, e' becomes the new estimate for e^*.

Input Augmentation. One challenge in the algorithm is that under the policy e, the agent may encounter histories that are not possible under e_{NN}, or any of the programs encountered in previous iterations of the search. For example, while searching for a steering controller, we may arrive at a program that, under certain conditions, steers the car into a wall, an illegal behavior that the neural policy does not exhibit. Such histories would be irrelevant to the distance between e_{NN} and e if the set $\mathcal{H}$ were constructed ahead of time

by simulating e_{NN}, and never updated. This would be unfortunate as these are precisely the inputs on which the programmatic policy needs guidance.

Our solution to this problem is *input augmentation*, or periodic updates to the set $\mathcal{H}$. More precisely, after a certain number of search steps for a fixed set $\mathcal{H}$, and after choosing the best available synthesized program for this set, we sample a set of additional histories by simulating the current programmatic policy, and add these samples to $\mathcal{H}$.

Algorithm 4 Neurally Directed Program Search

Input: Neural policy e_{NN}, POMDP M, sketch $\mathcal{S}$
$\mathcal{H} \leftarrow$ `create_histories`(e_{NN}, M)
$e \leftarrow$ `initialize`$(e_{NN}, \mathcal{H}, M, \mathcal{S})$
$R \leftarrow$ `collect_reward`(e, M)
repeat
  $(e', R') \leftarrow (e, R)$
  $\mathcal{H} \leftarrow$ `update_histories`$(e, e_{NN}, M, \mathcal{H})$
  $\mathcal{E} \leftarrow$ `neighborhood_pool`(e)
  $e \leftarrow \text{argmin}_{e' \in \mathcal{E}} \sum_{h \in \mathcal{H}} \|e'(h) - e_{NN}(h)\|$
  $R \leftarrow$ `collect_reward`(e, M)
until $R' \geq R$
Output: e'

We show pseudocode for NDPS in Algorithm 4. The inputs to the algorithm are a POMDP M, a neural policy e_{NN} for M that serves as an oracle, and a sketch $\mathcal{S}$. The algorithm first samples a set of histories of e_{NN} using the procedure `create_histories`. Next it uses the routine `initialize` to generate the program that is the starting point of the policy search. Then the procedure `collect_reward` calculates the expected aggregate reward $R(e)$ (described in Section 3.2), by simulating the program in the POMDP.

From this point on, NDPS iteratively updates its estimate e of the target program, as well as its estimate $\mathcal{H}$ of the set of interesting inputs used for distance computation. To do the former, NDPS uses the procedure `neighborhood_pool` to generate a space of programs that are structurally similar to e, then finds the program in this space that minimizes distance from e_{NN}. The latter task is done by the routine `update_histories`, which uses some heuristics to pick interesting inputs in the trajectory of the learned program and then obtains the corresponding actions from the oracle for those inputs. Intuitively, the distribution of the states in $\mathcal{H}$ is initially the oracle's state distribution, which subsequently gets augmented with the states visited by the learned program e, but the action always comes from the oracle. This process goes on until the iterative search fails to improve the estimated reward R of e.

The subroutines used in the above description can be implemented in many ways. Now we elaborate on our implementation of the important subroutines of NDPS.

The optimization step. The search for a program e' at minimal distance from the neural oracle can be implemented in many ways. The approach we use has two steps. First, we enumerate a set of *program templates* — numerically parameterized programs — that are structurally similar to e and are permitted by the sketch $\mathcal{S}$, giving priority to shorter templates. Next, we find optimal parameters for the enumerated templates using Bayesian optimization [85].

The initialization step. The performance of NDPS turns out to be quite sensitive to the choice of the program that is the starting point of the search. Our implementation of the initialization routine `initialize` starts by enumerating a pool of candidate program templates that are permitted by $\mathcal{S}$, giving priority to shorter templates. Next, the histories in $\mathcal{H}$ are used to optimize the parameter choices (using Bayesian optimization) and to create a list of the best candidate programs, based on how well the programs imitate the actions of the oracle. Finally, `initialize` simulates the programs in the POMDP and returns the program that achieves the highest reward.

The template generation. The `initialize` routine is given a sketch $\mathcal{S}$, in a context-free grammar (CFG) notation, from which it generates a candidate program templates in two steps. First, syntactically correct 'sentences' are randomly generated from the CFG. Next, the holes for the sensors (x_i) are filled by randomly sampling from the set of available sensors. This process is repeated to create a pool of distinct program templates. There is an implicit prior in the template generation that encourages the production of compact programs ie. programs with a small number of parameters.

5.3 Distillation into Decision Trees

The PROPEL framework provides the user the flexibility to choose a DSL that finds the best balance of expressiveness vs program search efficiency. More expressive languages create larger programmatic classes, thus making

the projection step via program synthesis more computationally expensive. In contrast, lower level representations, while easier to project into, provide less interpretable policies, that generalize poorly, are harder to verify, and lack mechanisms to include partial domain knowledge.

Among recent state-of-the-art approaches to learning programmatic policies, VIPER [11] learns decision tree-based policies. Both NDPS and VIPER rely on imitating a fixed (pre-trained) neural policy oracle, and can be viewed as degenerate versions of PROPEL that only run Lines 4-6 in Algorithm 1. In our experimental evaluations we consider a version of the PROPEL framework where the programmatic class is the space of all regression trees, similar to VIPER but in continuous control environments. Comparisons of this class with policies generated in the CFG described above demonstrate the trade-offs between choosing a more expressive policy class vs a comparatively low-level representation.

Lower-level representations like regression trees can be viewed as degenerate versions of the DSL described in Figure 4.2 because they lack the ability to include used defined "library functions" in the learnt programs, and are restricted to 'if-then-else' statements and output actions. However, this defines a more restricted program search space and it is therefore less computationally expensive to apply the projection operation. For the empirical evaluations we use the CART [17] algorithm to learn regression trees that mimic a neural oracle, with input augmentation performed as in the NDPS algorithm. The theoretical analysis of PROPEL is not strictly dependent on the structure of

the programmatic policy class, as long as the the assumptions detailed in Section 6.3 are valid.

Chapter 6

Theoretical Analysis

In this chapter we present a theoretical analysis of the PROPEL framework. We start by viewing PROPEL through the lens of online learning in function space, independent of the specific parametric representation. This start point yields a convergence analysis of Algorithm 1 under generic approximation errors. We then analyze the issues of policy class representation in Sections 6.2.2 and 6.2.3, and connect Algorithms 2 & 3 with the overall performance, under some simplifying assumptions. In particular, Section 6.2.3 characterizes the update error in a possibly non-differentiable setting.

6.1 Motivation

In this section we discuss some of the key terms used in the theoretical analysis presented in this chapter, and their connections to the practical implementations discussed in Chapter 7. As we will see later, it is sometimes hard to verify if all the assumptions made for the theoretical analysis hold for a given domain and programmatic class. However, despite this gap between theory and practice, the analysis presented in this chapter can provide important insights and help inform design choices when applying the PROPEL framework

to novel environments and applications.

The theoretical analysis in this chapter is devoted to two key properties of the PROPEL framework. The first provides an expected regret bound for the programmatic policies returned by Algorithm 1, and the second studies the end-to-end learning performance from finite samples. The expected regret bound provides an upper bound of the expected loss of the learnt policies from the loss of the optimal programmatic policy, in terms of the projection error and bias-variance of the gradient estimates. The finite sample analysis provides insight in the relative trade-offs of spending effort in obtaining more accurate projections versus more reliable gradient estimates, the two main computationally expensive components of the PROPEL framework. Taken together, these results provide insights into the convergence behavior of Algorithm 1.

The theoretical analysis is made under some simplifying assumptions about the policy spaces $\mathcal{H}, \Pi$,and $\mathcal{F}$ and the behavior of the loss function J. Here Π is the space of programmatic policies, $\mathcal{F}$ is the space of neural policies, and $\mathcal{H}$ is the space of all policies that can be formed by combining a neural and a programmatic policy. All these policy spaces can be embedded into an ambient policy space, which is the vector space of all functions from the set of States ($\mathcal{S}$) to the set of Actions ($\mathcal{A}$). Since we are primarily interested in applications to continuous control, we can view this as the function space from $\mathbb{R}^n$ to $\mathbb{R}^m$, for some n and m.

To analyze the behavior of the proposed algorithms, we consider functionals over the policies (which are functions from $\mathcal{S}$ to $\mathcal{A}$). A functional is

a rule that associates a number with a function, in our case this means it is a mapping from policies to the reals. A generalization of the concept of the derivative (for functions) is required to study the properties of the functionals. A standard choice for this generalization is the Fréchet functional gradient. This generalization (defined in Section 6.2) maps many of the properties of standard gradients to functional gradients, and provides the required framework to systematically study the variations of the underlying policy functions.

One of the key assumptions required for the theoretical analysis is that J is convex and Fréchet differentiable on $\mathcal{H}$. For an arbitrary environment and programmatic class, it may not be possible to verify this assumption. Furthermore, in the absence of this assumption, examples can be constructed that violate the theoretical guarantees presented in this chapter. For reasonably expressive programmatic classes, and the type of loss functions that are required for most environments, we believe that this assumption is usually true. Even in cases where the assumption cannot be verified, the insights from the theoretical results can be used to guide design choices in the implementation.

The assumptions that J is also differentiable on the restricted subspace $\mathcal{F}$ and that $\mathcal{F}$ is dense in $\mathcal{H}$, is somewhat easier to justify, since neural networks are universal function approximators. However, for any specific implementation of the PROPEL framework, we will usually need to set an upper bound on the depth and width of the neural networks. This means that in practice this assumption may be violated if our neural policy class is not sufficiently large. Once again, while the specific theoretical guarantees might not hold

for a particular implementation, the design decisions could benefit from the insights provided by the theorems.

6.2 Background and Results

We consider Π and $\mathcal{F}$ to be subspaces of an ambient policy space $\mathcal{U}$, which is a vector space equipped with inner product $\langle\cdot,\cdot\rangle$, induced norm $\|u\| = \sqrt{\langle u, u\rangle}$, dual norm $\|v\|_* = \sup\{\langle v, u\rangle | \, \|u\| \leq 1\}$, and standard scaling & addition: $(au + bv)(s) = au(s) + bv(s)$ for $a, b \in \mathbb{R}$ and $u, v \in \mathcal{U}$. The cost functional of a policy u is $J(u) = \int_{\mathcal{S}} c(s, u(s)) d\mu^u(s)$, where μ^u is the distribution of states induced by u. The joint policy class is $\mathcal{H} = \Pi \oplus \mathcal{F}$, by $\mathcal{H} = \{\pi + f | \forall \pi \in \Pi, f \in \mathcal{F}\}$.[1] Note that $\mathcal{H}$ is a subspace of $\mathcal{U}$, and inherits its vector space properties. Without affecting the analysis, we simply equate $\mathcal{U} \equiv \mathcal{H}$ for the remainder of the Chapter.

We assume that J is convex in $\mathcal{H}$, which implies that subgradient $\partial J(h)$ exists (with respect to $\mathcal{H}$) [12]. Where J is differentiable, we utilize the notion of a Fréchet gradient. Recall that a bounded linear operator $\nabla : \mathcal{H} \mapsto \mathcal{H}$ is called a Fréchet functional gradient of J at $h \in \mathcal{H}$ if $\lim\limits_{\|g\| \to 0} \frac{J(h+g) - J(h) - \langle \nabla J(h), g\rangle}{\|g\|} = 0$. By default, ∇ (or $\nabla_{\mathcal{H}}$ for emphasis) denotes the gradient with respect to $\mathcal{H}$, whereas $\nabla_{\mathcal{F}}$ defines the gradient in the restricted subspace $\mathcal{F}$.

[1] The operator $\oplus$ is not a direct sum, since Π and $\mathcal{F}$ are not orthogonal.

6.2.1 PROPEL as (Approximate) Functional Mirror Descent

For our analysis, PROPEL can be viewed as approximating mirror descent in (infinite-dimensional) function space over a convex set $\Pi \subset \mathcal{H}$.[2] Similar to the finite-dimensional setting [69], we choose a strongly convex and smooth functional regularizer R to be the mirror map. From the approximate mirror descent perspective, for each iteration t:

1. Obtain a noisy gradient estimate: $\widehat{\nabla}_{t-1} \approx \nabla J(\pi_{t-1})$
2. $\text{UPDATE}_{\mathcal{H}}(\pi)$ in $\mathcal{H}$ space: $\nabla R(h_t) = \nabla R(\pi_{t-1}) - \eta \widehat{\nabla}_{t-1}$

   (Note $\text{UPDATE}_{\mathcal{H}} \neq \text{UPDATE}_{\mathcal{F}}$*)*
3. Obtain approximate projection:

   $\pi_t = \text{PROJECT}_{\Pi}^{R}(h_t) \approx \text{argmin}_{\pi \in \Pi} D_R(\pi, h_t)$

$D_R(u, v) = R(u) - R(v) - \langle \nabla R(u), u - v \rangle$ is a Bregman divergence. Taking $R(h) = \frac{1}{2} \|h\|^2$ will recover projected functional gradient descent in $L2$-space. Here UPDATE becomes $h_t = \pi_{t-1} - \eta \widehat{\nabla} J(\pi_{t-1})$, and PROJECT solves for $\text{argmin}_{\pi \in \Pi} \|\pi - h_t\|^2$. While we mainly focus on this choice of R in our experiments, note that other selections of R lead to different UPDATE and PROJECT operators (e.g., minimizing KL divergence if R is negative entropy).

The functional mirror descent scheme above may encounter two additional sources of error compared to standard mirror descent [69]. First, in the stochastic setting (also called bandit feedback [36]), the gradient estimate $\widehat{\nabla}_t$

[2] Π can be convexified by considering *randomized* policies, as stochastic combinations of $\pi \in \Pi$ (cf. [58]).

may be biased, in addition to having high variance. One potential source of bias is the gap between $\textsc{Update}_{\mathcal{H}}$ and $\textsc{Update}_{\mathcal{F}}$. Second, the $\textsc{Project}$ step may be inexact. We start by analyzing the behavior of $\textsc{Propel}$ under generic bias, variance, and projection errors, before discussing the implications of approximating $\textsc{Update}_{\mathcal{H}}$ and $\textsc{Project}_{\Pi}$ by Algs. 2 & 3, respectively. Let the bias be bounded by β, i.e., $\left\|\mathbb{E}[\widehat{\nabla}_t|\pi_t] - \nabla J(\pi_t)\right\|_* \leq \beta$ almost surely. Similarly let the variance of the gradient estimate be bounded by σ^2, and the projection error norm $\|\pi_t - \pi_t^*\| \leq \epsilon$. We state the expected regret bound below; more details and a proof appear in Section 6.3.

Theorem 6.2.1 (Expected regret bound under gradient estimation and projection errors)**.** *Let $\pi_1, \ldots, \pi_T$ be a sequence of programmatic policies returned by Algorithm 1, and π^* be the optimal programmatic policy. Choosing learning rate $\eta = \sqrt{\frac{1}{\sigma^2}(\frac{1}{T} + \epsilon)}$, we have the expected regret over T iterations:*

$$\mathbb{E}\left[\frac{1}{T}\sum_{t=1}^{T} J(\pi_t)\right] - J(\pi^*) = O\left(\sigma\sqrt{\frac{1}{T} + \epsilon} + \beta\right). \tag{6.1}$$

The result shows that error ϵ from $\textsc{Project}$ and the bias β do not accumulate and simply contribute an additive term on the expected regret.[3] The effect of variance of gradient estimate decreases at a $\sqrt{1/T}$ rate. Note that this regret bound is agnostic to the specific $\textsc{Update}$ and $\textsc{Project}$ operations, and can be applied more generically beyond the specific algorithmic choices used in this thesis.

[3] Other mirror descent-style analyses, such as in [88], lead to accumulation of errors over the rounds of learning T. One key difference is that we are leveraging the assumption of convexity of J in the (infinite-dimensional) function space representation.

6.2.2 Finite-Sample Analysis under Vanilla Policy Gradient Update and DAgger Projection

Next, we show how certain instantiations of UPDATE and PROJECT affect the magnitude of errors and influence end-to-end learning performance from finite samples, under some simplifying assumptions on the UPDATE step. For this analysis, we simplify Alg. 2 into the case $\text{UPDATE}_{\mathcal{F}} \equiv \text{UPDATE}_{\mathcal{H}}$. In particular, we assume programmatic policies in Π to be parameterized by a vector $\theta \in \mathbb{R}^k$, and π is differentiable in θ (e.g., we can view $\Pi \subset \mathcal{F}$ where $\mathcal{F}$ is parameterized in $\mathbb{R}^k$). We further assume the trajectory roll-out is performed in an exploratory manner, where action is taken uniformly random over finite set of A actions, thus enabling the bound on the bias of gradient estimates via Bernstein's inequality. The PROJECT step is consistent with Alg. 3, i.e., using DAgger [74] under convex imitation loss, such as ℓ_2 loss. We have the following high-probability guarantee:

Theorem 6.2.2 (Finite-sample guarantee)**.** *At each iteration, we perform vanilla policy gradient estimate of π (over $\mathcal{H}$) using m trajectories and, use DAgger algorithm to collect M roll-outs for the imitation learning projection. Setting the learning rate $\eta = \sqrt{\frac{1}{\sigma^2}\left(\frac{1}{T} + \frac{H}{M} + \sqrt{\frac{\log(T/\delta)}{M}}\right)}$, after T rounds of the algorithm, we have that:*

$$\frac{1}{T}\sum_{t=1}^{T} J(\pi_t) - J(\pi^*) \leq O\left(\sigma\sqrt{\frac{1}{T} + \frac{H}{M} + \sqrt{\frac{\log(T/\delta)}{M}}}\right) + O\left(\sigma\sqrt{\frac{\log(Tk/\delta)}{m}} + \frac{AH\log(Tk/\delta)}{m}\right)$$

holds with probability at least $1-\delta$*, with* H *being the task horizon,* A *the cardinality of action space,* σ^2 *the variance of policy gradient estimates, and* k *the dimension* Π*'s parameterization.*

The expanded result and proof are included in Section 6.3. The proof leverages previous analysis from DAgger [75] and the finite sample analysis of vanilla policy gradient algorithm [48]. The finite-sample regret bound scales linearly with the standard deviation σ of the gradient estimate, while the bias, which is the very last component of the RHS, scales linearly with the task horizon H. Note that the standard deviation σ can be exponential in task horizon H in the worst case [48], and so it is important to have practical implementation strategies to reduce the variance of the UPDATE operation. While conducted in a stylized setting, this analysis provides insight in the relative trade-offs of spending effort in obtaining more accurate projections versus more reliable gradient estimates.

6.2.3 Closing the gap between the gradient estimates

Our functional mirror descent analysis rests on taking gradients in $\mathcal{H}$: $\text{UPDATE}_{\mathcal{H}}(\pi)$ involves estimating $\nabla_{\mathcal{H}} J(\pi)$ in the $\mathcal{H}$ space. On the other hand,

Algorithm 2 performs $\textsc{Update}_{\mathcal{F}}(\pi)$ only in the neural policy space $\mathcal{F}$. In either case, although $J(\pi)$ may be differentiable in the non-parametric ambient policy space, it may not be possible to obtain a differentiable parametric programmatic representation in Π. In this section, we discuss theoretical motivations to addressing a practical issue: *How do we define and approximate the gradient $\nabla_{\mathcal{H}} J(\pi)$ under a parametric representation?* To our knowledge, we are the first to consider such a theoretical question for reinforcement learning.

Defining a consistent approximation of $\nabla_{\mathcal{H}} J(\pi)$. The idea in $\textsc{Update}_{\mathcal{F}}(\pi)$ (Line 8 of Alg. 1) is to approximate $\nabla_{\mathcal{H}} J(\pi)$ by $\nabla_{\mathcal{F}} J(f)$, which has a differentiable representation, at some f close to π (under the norm). Under appropriate conditions on $\mathcal{F}$, we show that this approximation is valid.

Proposition 6.2.3. *Assume that (i) J is Fréchet differentiable on $\mathcal{H}$, (ii) J is also differentiable on the restricted subspace $\mathcal{F}$, and (iii) $\mathcal{F}$ is dense in $\mathcal{H}$ (i.e., the closure $\overline{\mathcal{F}} = \mathcal{H}$). Then for any fixed policy $\pi \in \Pi$, define a sequence of policies $f_k \in \mathcal{F}$, $k = 1, 2, \ldots$), that converges to π: $\lim_{k\to\infty} \|f_k - \pi\| = 0$. We then have $\lim_{k\to\infty} \|\nabla_{\mathcal{F}} J(f_k) - \nabla_{\mathcal{H}} J(\pi)\|_* = 0$.*

Since the Fréchet gradient is unique in the ambient space $\mathcal{H}$, $\forall k$ we have $\nabla_{\mathcal{H}} J(f_k) = \nabla_{\mathcal{F}} J(f_k) \to \nabla_{\mathcal{H}} J(\pi)$ as $k \to \infty$ (by Proposition 6.2.3). We thus have an asymptotically unbiased approximation of $\nabla_{\mathcal{H}} J(\pi)$ via differentiable space $\mathcal{F}$ as: $\nabla_{\mathcal{F}} J(\pi) \triangleq \nabla_{\mathcal{H}} J(\pi) \triangleq \lim_{k\to\infty} \nabla_{\mathcal{F}} J(f_k)$.[4] Connecting to the result

[4] We do not assume $J(\pi)$ to be differentiable when restricting to the policy subspace Π, i.e., $\nabla_{\Pi} J(\pi)$ may not exist under policy parameterization of Π.

from Theorem 6.2.1, let σ^2 be an upper bound on the policy gradient estimates in the *neural policy class* $\mathcal{F}$, under an asymptotically unbiased approximation of $\nabla_{\mathcal{H}} J(\pi)$, the expected regret bound becomes $\mathbb{E}\left[\frac{1}{T}\sum_{t=1}^{T} J(\pi_t)\right] - J(\pi^*) = O\left(\sigma\sqrt{\frac{1}{T}+\epsilon}\right)$.

Bias-variance considerations of $\textsc{Update}_{\mathcal{F}}(\pi)$ To further theoretically motivate a practical strategy for $\textsc{Update}_{\mathcal{F}}(\pi)$ in Algorithm 2, we utilize an equivalent proximal perspective of mirror descent [13], where $\textsc{Update}_{\mathcal{H}}(\pi)$ is equivalent to solving for $h' = \operatorname{argmin}_{h\in\mathcal{H}} \eta\langle\nabla_{\mathcal{H}} J(\pi), h\rangle + D_R(h, \pi)$.

Proposition 6.2.4 (Minimizing a relaxed objective)**.** *For a fixed programmatic policy π, with sufficiently small constant $\lambda \in (0,1)$, we have that*

$$\min_{h\in\mathcal{H}} \eta\langle\nabla_{\mathcal{H}} J(\pi), h)\rangle + D_R(h,\pi) \leq \min_{f\in\mathcal{F}} J\big(\pi+\lambda f\big) - J(\pi) + \langle\nabla J(\pi), \pi\rangle \quad (6.2)$$

Thus, a relaxed $\textsc{Update}_{\mathcal{H}}$ step is obtained by minimizing the RHS of (6.2), i.e., minimizing $J(\pi+\lambda f)$ over $f \in \mathcal{F}$. Each gradient descent update step is now $f' = f - \eta\lambda\nabla_{\mathcal{F}} J(\pi_t + \lambda f)$, corresponding to Line 5 of Algorithm 2. For fixed π and small λ, this relaxed optimization problem becomes regularized policy optimization over $\mathcal{F}$, which is significantly easier. Functional regularization in policy space around a fixed prior controller π has demonstrated significant reduction in the variance of gradient estimate [25], at the expense of some bias. The below expected regret bound summarizes the impact of this increased bias and reduced variance, with details included in Section 6.3.

Proposition 6.2.5 (Bias-variance characterization of $\textsc{Update}_{\mathcal{F}}$). *Assuming $J(h)$ is L-strongly smooth over $\mathcal{H}$, i.e., $\nabla_{\mathcal{H}} J(h)$ is L-Lipschitz continuous, approximating* $\textsc{Update}_{\mathcal{H}}$ *by* $\textsc{Update}_{F}$ *per Alg. 2 leads to the expected regret bound:* $\mathbb{E}\left[\frac{1}{T}\sum_{t=1}^{T} J(\pi_t)\right] - J(\pi^*) = O\left(\lambda\sigma\sqrt{\frac{1}{T}+\epsilon} + \lambda^2 L^2\right)$.

Compared to the idealized unbiased approximation in Proposition 6.2.3, the introduced bias here is related to the inherent smoothness property of cost functional $J(h)$ over the joint policy class $\mathcal{H}$, i.e., how close $J(\pi + \lambda f)$ is to its linear under-approximation $J(\pi) + \langle \nabla_{\mathcal{H}} J(\pi), \lambda f \rangle$ around π.

6.3 Details and Proofs

We formally define an ambient control policy space $\mathcal{U}$ to be a vector space equipped with inner product $\langle \cdot, \cdot \rangle : \mathcal{U} \times \mathcal{U} \mapsto \mathbb{R}$, which induces a norm $\|u\| = \sqrt{\langle u, u \rangle}$, and its dual norm defined as $\|v\|_* = \sup\{\langle v, u \rangle | \, \|u\| \leq 1\}$. While multiple ways to define the inner product exist, for concreteness we can think of the example of square-integrable stationary policies with $\langle u, v \rangle = \int_{\mathcal{S}} u(s)v(s)ds$. The addition operator $+$ between two policies $u, v \in \mathcal{U}$ is defined as $(u + v)(s) = u(s) + v(s)$ for all state $s \in \mathcal{S}$. Scaling $\lambda u + \kappa v$ is defined similarly for scalar λ, κ.

The cost functional of a control policy u is defined as

$$J(u) = \int_0^\infty c(s(\tau), u(\tau))d\tau, \quad \text{or} \quad J(u) = \int_{\mathcal{S}} c(s, u(s))d\mu^u(s),$$

where μ^u is the distribution of states induced by policy u. This latter example is equivalent to the standard notion of value function in reinforcement learning.

Separate from the parametric representation issues, both programmatic policy class Π and neural policy class $\mathcal{F}$, and by extension - the joint policy class $\mathcal{H}$, are considered to live in the ambient vector space $\mathcal{U}$. We thus have a common and well-defined notion of distance between policies from different classes.

We make an important distinction between differentiability of $J(h)$ in the ambient policy space (non-parametric), versus differentiability in parameterization (parametric). For example, if Π is a class of decision-tree based policy, policies in Π may not be differentiable under representation. However, policies $\pi \in \Pi$ might still be differentiable when considered as points in the ambient vector space $\mathcal{U}$.

We will use the following standard notion of gradient and differentiability from functional analysis:

Definition 6.3.1 (Subgradients)**.** The subgradient of J at h, denoted $\partial J(h)$, is the non-empty set $\{g \in \mathcal{H} | \forall j \in \mathcal{H} : \langle j - h, g\rangle + J(h) \leq J(j)\}$

Definition 6.3.2 (Fréchet gradient)**.** A bounded linear operator $\nabla : \mathcal{H} \mapsto \mathcal{H}$ is called Fréchet functional gradient of J at $h \in \mathcal{H}$ if $\lim\limits_{\|g\| \to 0} \frac{J(h+g) - J(h) - \langle \nabla J(h), g\rangle}{\|g\|} = 0$

The notions of convexity, smoothness and Bregman divergence are analogous to finite-dimensional setting:

Definition 6.3.3 (Strong convexity)**.** A differentiable function R is $\alpha-$strongly convex w.r.t norm $\|\cdot\|$ if $R(y) \geq R(x) + \langle \nabla R(x), y - x\rangle + \frac{\alpha}{2} \|y - x\|^2$

Definition 6.3.4 (Lipschitz continuous gradient smoothness)**.** A differentiable function R is L_R-strongly smooth w.r.t norm $\|\cdot\|$ if

$$\|\nabla R(x) - \nabla R(y)\|_* \leq L_R \|x - y\|.$$

Definition 6.3.5 (Bregman Divergence)**.** For a strongly convex regularizer R, $D_R(x, y) = R(x) - R(y) - \langle \nabla R(y), x - y \rangle$ is the Bregman divergence between x and y (not necessarily symmetric).

The following standard result for Bregman divergence will be useful:

Lemma 6.3.1. *[13] For all x, y, z we have the identity*

$$\langle \nabla R(x) - \nabla R(y), x - z \rangle = D_R(x, y) + D_R(z, x) - D_R(z, y).$$

Since Bregman divergence is non-negative, a consequence of this identity is that

$$D_R(z, x) - D_R(z, y) \leq \langle \nabla R(x) - \nabla R(y), z - x \rangle.$$

6.3.1 Expected Regret Bound under Noisy Policy Gradient Estimates and Projection Errors

In this section, we show regret bound for the performance of the sequence of returned programs $\pi_1, \dots, \pi_T$ of the algorithm. The analysis here is agnostic to the particular implementation of algorithm 2 and algorithm 3.

Let R be a $\alpha-$strongly convex and L_R-smooth functional with respect to norm $\|\cdot\|$ on $\mathcal{H}$. The steps from algorithm 1 can be described as follows.

- Initialize $\pi_0 \in \Pi$. For each iteration t:

1. Obtain a noisy estimate of the gradient $\widehat{\nabla} J(\pi_{t-1}) \approx \nabla J(\pi_{t-1})$
2. Update in the $\mathcal{H}$ space: $\nabla R(h_t) = \nabla R(\pi_{t-1}) - \eta \widehat{\nabla} J(\pi_{t-1})$
3. Obtain approximate projection π_t :

$$\pi_t = \text{PROJECT}_{\pi}^{R}(h_t) \approx \underset{\pi \in \Pi}{\operatorname{argmin}}\, D_R(\pi, h_t).$$

This procedure is an approximate functional mirror descent scheme under bandit feedback. We will develop the following result.

In the statement below, D is the diameter on Π with respect to defined norm $\|\cdot\|$ (i.e., $D = \sup \|\pi - \pi'\|$). L_J is the Lipschitz constant of the functional J on $\mathcal{H}$. β, σ^2 are the bound on the bias and variance of the gradient estimate at each iteration, respectively. α and L_R are the strongly convex and smooth coefficients of the functional regularizer R. Finally, ϵ is the bound on the projection error with respect to the same norm $\|\cdot\|$.

Theorem 6.3.2 (Regret bound of returned policies)**.** *Let $\pi_1, \ldots, \pi_T$ be a sequence of programmatic policies returned by algorithm 1 and π^* be the optimal programmatic policy. We have the expected regret bound:*

$$\mathbb{E}\left[\frac{1}{T}\sum_{t=1}^{T} J(\pi_t)\right] - J(\pi^*) \le \frac{L_R D^2}{\eta T} + \frac{\epsilon L_R D}{\eta} + \frac{\eta(\sigma^2 + L_J^2)}{\alpha} + \beta D$$

In particular, choosing the learning rate $\eta = \sqrt{\frac{\frac{1}{T}+\epsilon}{\sigma^2}}$, the expected regret is simplified into:

$$\mathbb{E}\left[\frac{1}{T}\sum_{t=1}^{T} J(\pi_t)\right] - J(\pi^*) = O\left(\sigma\sqrt{\frac{1}{T} + \epsilon} + \beta\right) \tag{6.3}$$

Proof. At each round t, let $\overline{\nabla}_t = \mathbf{E}[\widehat{\nabla}_t|\pi_t]$ be the conditional expectation of the gradient estimate. We will use the shorthand notation $\nabla_t = \nabla J(\pi_t)$. Denote the upper-bound on the bias of the estimate by β_t, i.e., $\left\|\overline{\nabla}_t - \nabla_t\right\|_* \leq \beta_t$ almost surely. Denote the noise of the gradient estimate by $\xi_t = \overline{\nabla}_t - \widehat{\nabla}_t$, and $\sigma_t^2 = \mathbf{E}\left[\left\|\widehat{\nabla}_t - \overline{\nabla}_t\right\|_*^2\right]$ is the variance of gradient estimate $\widehat{\nabla}_t$.

The projection operator is $\epsilon-$approximate in the sense that

$$\left\|\pi_t - \textsc{Project}_\Pi^R(f_t)\right\| = \left\|\widehat{\textsc{Project}}_\Pi^R(h_t) - \textsc{Project}_\Pi^R(h_t)\right\| \leq \epsilon$$

with some constant ϵ, which reflects the statistical error of the imitation learning procedure. This projection error in general is independent of the choice of function classes Π and $\mathcal{F}$. We will use the shorthand notation $\pi_t^* = \textsc{Project}_\Pi^R(f_t)$ for the true Bregman projection of h_t onto Π.

Due to convexity of J over the space $\mathcal{H}$ (which includes Π), we have for all $\pi \in \Pi$:

$$J(\pi_t) - J(\pi) \leq \langle \nabla_t, \pi_t - \pi \rangle$$

We proceed to bound the RHS, starting with bounding the inner product where

the actual gradient is replaced by the estimated gradient.

$$\langle \widehat{\nabla}_t, \pi_t - \pi \rangle = \frac{1}{\eta_t} \langle \nabla R(\pi_t) - \nabla R(h_{t+1}), \pi_t - \pi \rangle \tag{6.4}$$

$$= \frac{1}{\eta_t} \big(D_R(\pi, \pi_t) - D_R(\pi, h_{t+1}) + D_R(\pi_t, h_{t+1}) \big) \tag{6.5}$$

$$\leq \frac{1}{\eta_t} \big(D_R(\pi, \pi_t) - D_R(\pi, \pi^*_{t+1}) - D_R(\pi^*_{t+1}, h_{t+1}) + D_R(\pi_t, h_{t+1}) \big) \tag{6.6}$$

$$= \frac{1}{\eta_t} \big(\underbrace{D_R(\pi, \pi_t) - D_R(\pi, \pi_{t+1})}_{\text{telescoping}} + \underbrace{D_R(\pi, \pi_{t+1}) - D_R(\pi, \pi^*_{t+1})}_{\text{projection error}} \tag{6.7}$$

$$\underbrace{- D_R(\pi^*_{t+1}, h_{t+1}) + D_R(\pi_t, h_{t+1})}_{\text{relative improvement}} \big) \tag{6.8}$$

Equation (6.4) is due to the gradient update rule in $\mathcal{F}$ space. Equation (6.5) is derived from definition of Bregman divergence. Equation (6.6) is due to the generalized Pythagorean theorem of Bregman projection

$$D_R(x, y) \geq D_R(x, \textsc{Project}^R_\Pi(x)) + D_R(\textsc{Project}^R_\Pi(x), y).$$

The RHS of equation (6.6) are decomposed into three components that will be bounded separately.

Bounding projection error. By lemma (6.3.1) we have

$$D_R(\pi, \pi_{t+1}) - D_R(\pi, \pi^*_{t+1}) \leq \langle \nabla R(\pi_{t+1}) - \nabla R(\pi^*_{t+1}), \pi - \pi_{t+1} \rangle \tag{6.9}$$

$$\leq \left\| \nabla R(\pi_{t+1}) - \nabla R(\pi^*_{t+1}) \right\| \left\| \pi - \pi_{t+1} \right\|_* \tag{6.10}$$

$$\leq L_R \left\| \pi_{t+1} - \pi^*_{t+1} \right\| D \leq \epsilon L_R D \tag{6.11}$$

Equation (6.10) is due to Cauchy–Schwarz. Equation (6.11) is due to Lipschitz smoothness of ∇R and definition of $\epsilon-$approximate projection.

Bounding relative improvement. This follows standard argument from analysis of mirror descent algorithm.

$$D_R(\pi_t, h_{t+1}) - D_R(\pi^*_{t+1}, h_{t+1}) = R(\pi_t) - R(\pi^*_{t+1}) + \langle \nabla R(h_{t+1}), \pi^*_{t+1} - \pi_t \rangle \tag{6.12}$$

$$\leq \langle \nabla R(\pi_t), \pi_t - \pi^*_{t+1} \rangle - \frac{\alpha}{2} \left\| \pi^*_{t+1} - \pi_t \right\|_*^2 + \langle \nabla R(h_{t+1}), \pi^*_{t+1} - \pi_t \rangle \tag{6.13}$$

$$= -\eta_t \langle \widehat{\nabla}_t, \pi^*_{t+1} - \pi_t \rangle - \frac{\alpha}{2} \left\| \pi^*_{t+1} - \pi_t \right\|^2 \tag{6.14}$$

$$\leq \frac{\eta_t^2}{2\alpha} \left\| \widehat{\nabla}_t \right\|_*^2 \leq \frac{\eta_t^2}{\alpha} (\sigma_t^2 + L_J^2) \tag{6.15}$$

Equation (6.13) is from the $\alpha-$strong convexity property of regularizer R. Equation (6.14) is by definition of the gradient update. Combining the bounds on the three components and taking expectation, we thus have

$$\mathbb{E}\left[\langle \widehat{\nabla}_t, \pi_t - \pi \rangle \right] \leq \frac{1}{\eta_t} \left(D_R(\pi, \pi_t) - D_R(\pi, \pi_{t+1}) + \epsilon L_R D + \frac{\eta_t^2}{\alpha} (\sigma_t^2 + L_J^2) \right) \tag{6.16}$$

Next, the difference between estimated gradient $\widehat{\nabla}_t$ and actual gradient ∇_t factors into the bound via Cauchy-Schwarz:

$$\mathbb{E}\left[\langle \nabla_t - \widehat{\nabla}_t, \pi_t - \pi \rangle \right] \leq \left\| \nabla_t - \mathbb{E}[\widehat{\nabla}_t] \right\|_* \left\| \pi_t - \pi \right\| \leq \beta_t D \tag{6.17}$$

The results can be deduced from equations (6.16) and (6.17).

Unbiased gradient estimates. For the case when the gradient estimate is unbiased, assume the variance of the noise of gradient estimates is bounded by σ^2, we have the expected regret bound for all $\pi \in \Pi$

$$\mathbb{E}\left[\frac{1}{T} \sum_{t=1}^{T} J(\pi_t) \right] - J(\pi) \leq \frac{L_R D^2}{\eta T} + \frac{\epsilon L_R D}{\eta} + \frac{\eta(\sigma^2 + L_J^2)}{\alpha} \tag{6.18}$$

here to clarify, L_R is the smoothness coefficient of regularizer R (i.e., the gradient of R is L_R-Lipschitz, L_J is Lipschitz constant of J, D is the diameter of Π under norm $\|\cdot\|$, σ^2 is the upper-bound on the variance of gradient estimates, and ϵ is the error from the projection procedure (i.e., imitation learning loss).

We can set learning rate $\eta = \sqrt{\frac{\frac{1}{T}+\epsilon}{\sigma^2}}$ to observe that the expected regret is bounded by $O(\sigma\sqrt{\frac{1}{T}+\epsilon})$.

Biased gradient estimates. Assume that the bias of gradient estimate at each round is upper-bounded by $\beta_t \leq \beta$. Similar to before, combining inequalities from (6.16) and (6.17), we have

$$\mathbb{E}\left[\frac{1}{T}\sum_{t=1}^{T} J(\pi_t)\right] - J(\pi) \leq \frac{L_R D^2}{\eta T} + \frac{\epsilon L_R D}{\eta} + \frac{\eta(\sigma^2 + L_J^2)}{\alpha} + \beta D \tag{6.19}$$

Similar to before, we can set learning rate $\eta = \sqrt{\frac{\frac{1}{T}+\epsilon}{\sigma^2}}$ to observe that on the expected regret is bounded by $O(\sigma\sqrt{\frac{1}{T}+\epsilon}+\beta)$. Compared to the bound on (6.18), in the biased case, the extra regret incurred per bound is simply a constant, and does not depend on T. □

6.3.2 Finite-Sample Analysis

In this section, we provide overall finite-sample analysis for PROPEL under some simplifying assumptions. We first consider the case where exact gradient estimate is available, before extending the result to the general case of noisy policy gradient update. Combining the two steps will give us the proof for the following statement:

Theorem 6.3.3 (Finite-sample guarantee)**.** *At each iteration, we perform vanilla policy gradient estimate of* π *(over* $\mathcal{H}$*) using* m *trajectories and use DAgger algorithm to collect* M *roll-outs. Setting the learning rate*

$$\eta = \sqrt{\frac{1}{\sigma^2}\left(\frac{1}{T} + \frac{H}{M} + \sqrt{\frac{\log(T/\delta)}{M}}\right)},$$

after T *rounds of the algorithm, we have that*

$$\frac{1}{T}\sum_{t=1}^{T} J(\pi_t) - J(\pi^*) \leq O\left(\sigma\sqrt{\frac{1}{T} + \frac{H}{M} + \sqrt{\frac{\log(T/\delta)}{M}}}\right) + O\left(\sigma\sqrt{\frac{\log(Tk/\delta)}{m}} + \frac{AH\log(Tk/\delta)}{m}\right)$$

holds with probability at least $1-\delta$*, with* H *the task horizon,* A *the cardinality of action space,* σ^2 *the variance of policy gradient estimates, and* k *the dimension* Π*'s parameterization.*

Exact gradient estimate case. Assuming that the policy gradients can be calculated exactly, it is straight-forward to provide high-probability guarantee for the effect of the projection error. We start with the following result, adapted from [74] for the case of projection error bound. In this version of DAgger, we assume that we only collect a single *(state, expert action)* pair from each trajectory roll-out. Result is similar, with tighter bound, when multiple data points are collected along the trajectory.

Lemma 6.3.4 (Projection error bound from imitation learning procedure)**.** *Using DAgger as the imitation learning sub-routine for our* PROJECT *operator*

in algorithm 3, let M be the number of trajectories rolled-out for learning, and H be the horizon of the task. With probability at least $1-\delta$, we have

$$D_R(\pi, \pi^*) \leq \widetilde{O}(1/M) + \frac{2\ell_{max}(1+H)}{M} + \sqrt{\frac{2\ell_{max}\log(1/\delta))}{M}}$$

where π is the result of PROJECT*, π^* is the true Bregman projection of h onto Π, and ℓ_{max} is the maximum value of the imitation learning loss function $D_R(\cdot,\cdot)$*

The bound in lemma 6.3.4 is simpler than previous imitation learning results with cost information ([73, 74]. The reason is that the goal of the PROJECT operator is more modest. Since we only care about the distance between the empirical projection π and the true projection π^*, the loss objective in imitation learning is simplified (i.e., this is only a regret bound), and we can disregard how well policies in Π can imitate the expert h, as well as the performance of $J(\pi)$ relative to the true cost from the environment $J(h)$.

A consequence of this lemma is that for the number of trajectories at each round of imitation learning $M = O(\frac{\log 1/\delta}{\epsilon^2}) + O(\frac{H}{\epsilon})$, we have $D_R(\pi_t, \pi_t^*) \leq \epsilon$ with probability at least $1-\delta$. Applying union bound across T rounds of learning, we obtain the following guarantee (under no gradient estimation error)

Proposition 6.3.5 (Finite-sample Projection Error Bound)**.** *To simplify the presentation of the result, we consider L_R, D, L, α to be known constants. Using DAgger algorithm to collect $M = O\left(\frac{\log T/\delta}{\epsilon^2}\right) + O\left(\frac{H}{\epsilon}\right)$ roll-outs at each iteration,*

we have the following regret guarantee after T rounds of our main algorithm:

$$\frac{1}{T}\sum_{t=1}^{T} J(\pi_t) - J(\pi^*) \leq O\left(\frac{1}{\eta T} + \frac{\epsilon}{\eta} + \eta\right)$$

with probability at least $1-\delta$. Consequently, setting

$$\eta = \sqrt{\frac{1}{T} + \frac{H}{M} + \sqrt{\frac{\log(T/\delta)}{M}}},$$

we have that

$$\frac{1}{T}\sum_{t=1}^{T} J(\pi_t) - J(\pi^*) \leq O\left(\sqrt{\frac{1}{T} + \frac{H}{M} + \sqrt{\frac{\log(T/\delta)}{M}}}\right)$$

with probability at least $1-\delta$

Note that the dependence on the time horizon of the task is sub-linear. This is different from standard imitation learning regret bounds, which are often at least linear in the task horizon. The main reason is that our comparison benchmark π^* does live in the space Π, whereas for DAgger, the expert policy may not reside in the same space.

Noisy gradient estimate case. We now turn to the issue of estimating the gradient of $\nabla J(\pi)$. We make the following simplifying assumption about the gradient estimation:

- The π is parameterized by vector $\theta \in \mathbb{R}^k$ (such as a neural network). The parameterization is differentiable with respect to θ (Alternatively, we can view Π as a differentiable subspace of $\mathcal{F}$, in which case we have $\mathcal{H} = \mathcal{F}$)

- At each UPDATE loop, the policy is rolled out m times to collect the data, each trajectory has horizon length H

- For each visited state $s \sim d_h$, the policy takes a uniformly random action a. The action space is finite with cardinality A.

- The gradient ∇h_θ is bounded by B

The gradient estimate is performed consistent with a generic policy gradient scheme, i.e.,

$$\widehat{\nabla} J(\theta) = \frac{A}{m} \sum_{i=1}^{H} \sum_{j=1}^{m} \nabla \pi_\theta (a_i^j | s_i^j, \theta) \widehat{Q}_i^j$$

where $\widehat{Q}_i^j$ is the estimated cost-to-go [91].

Taking uniform random exploratory actions ensures that the samples are i.i.d. We can thus apply Bernstein's inequality to obtain the bound between estimated gradient and the true gradient. Indeed, with probability at least $1 - \delta$, we have that the following bound on the bias component-wise:

$$\left\| \widehat{\nabla} J(\theta) - \nabla J(\theta) \right\|_\infty \leq \beta \text{ when } m \geq \frac{(2\sigma^2 + 2AHB\frac{\beta}{3}) \log \frac{k}{\delta}}{\beta^2}$$

which leads to similar bound with respect to $\|\cdot\|_*$ (here we leverage the equivalence of norms in finite dimensional setting):

$$\left\| \nabla_t - \widehat{\nabla}_t \right\|_* \leq \beta \text{ when } m = O\left(\frac{(\sigma^2 + AHB\beta) \log \frac{k}{\delta}}{\beta^2} \right)$$

Applying union bound of this result over T rounds of learning, and combining with the result from proposition (6.3.5), we have the following finite-sample guarantee in the simplifying policy gradient update.

Proposition 6.3.6 (Finite-sample Guarantee under Noisy Gradient Updates and Projection Error). *At each iteration, we perform policy gradient estimate using* $m = O\left(\frac{(\sigma^2 + AHB\beta)\log\frac{Tk}{\delta}}{\beta^2}\right)$ *trajectories and use DAgger algorithm to collect* $M = O\left(\frac{\log T/\delta}{\epsilon^2}\right) + O\left(\frac{H}{\epsilon}\right)$ *roll-outs. Setting the learning rate*

$$\eta = \sqrt{\frac{1}{\sigma^2}(\frac{1}{T} + \frac{H}{M} + \sqrt{\frac{\log(T/\delta)}{M}})},$$

after T *rounds of the algorithm, we have that*

$$\frac{1}{T}\sum_{t=1}^{T} J(\pi_t) - J(\pi^*) \leq O\left(\sigma\sqrt{\frac{1}{T} + \frac{H}{M} + \sqrt{\frac{\log(T/\delta)}{M}}}\right) + \beta$$

with probability at least $1-\delta$.

Consequently, we also have the following regret bound:

$$\frac{1}{T}\sum_{t=1}^{T} J(\pi_t) - J(\pi^*) \leq O\left(\sigma\sqrt{\frac{1}{T} + \frac{H}{M} + \sqrt{\frac{\log(T/\delta)}{M}}}\right) + O\left(\sigma\sqrt{\frac{\log(Tk/\delta)}{m}} + \frac{AH\log(Tk/\delta)}{m}\right)$$

holds with probability at least $1-\delta$*, where again* H *is the task horizon,* A *is the cardinality of action space, and* k *is the dimension of function class* Π*'s parameterization.*

Proof. (For both proposition (6.3.6) and (6.3.5)). The results follow by taking the inequality from equation (6.19), and by solving for ϵ and β explicitly in terms of relevant quantities. Based on the specification of M and m, we

obtain the necessary precision for each round of learning in terms of number of trajectories:

$$\beta = O\left(\sigma\frac{\log(k/\delta)}{m} + \frac{AHB\log(k/\delta)}{m}\right)$$
$$\epsilon = O\left(\frac{H}{M} + \sqrt{\frac{\log(1/\delta)}{M}}\right)$$

Setting the learning rate $\eta = \sqrt{\frac{1}{\sigma^2}\left(\frac{1}{T}+\epsilon\right)}$ and rearranging the inequalities lead to the desired bounds. □

The regret bound depends on the variance σ^2 of the policy gradient estimates. It is well-known that vanilla policy gradient updates suffer from high variance. We instead use functional regularization technique, based on CORE-RL, in the practical implementation of our algorithm. The CORE-RL subroutine has been demonstrated to reduce the variance in policy gradient updates [25].

6.3.3 Defining a consistent approximation of the gradient

We are using the notion of Fréchet derivative to define gradient of differentiable functional. Note that while Gateaux derivative can also be utilized, Fréchet derivative ensures continuity of the gradient operator that would be useful for our analysis.

Definition 6.3.6 (Fréchet gradient)**.** A bounded linear operator $\nabla : \mathcal{H} \mapsto \mathcal{H}$ is called Fréchet functional gradient of J at $h \in \mathcal{H}$ if

$$\lim_{\|g\|\to 0} \frac{J(h+g) - J(h) - \langle \nabla J(h), g\rangle}{\|g\|} = 0.$$

We make the following assumption about $\mathcal{H}$ and $\mathcal{F}$. One interpretation of this assumption is that the space of policies Π and $\mathcal{F}$ that we consider have the property that a programmatic policy $\pi \in \Pi$ can be well-approximated by a large space of neural policies $f \in \mathcal{F}$.

Assumption 6.3.7. *J is Fréchet differentiable on $\mathcal{H}$. J is also differentiable on the restricted subspace $\mathcal{F}$. And $\mathcal{F}$ is dense in $\mathcal{H}$ (i.e., the closure $\overline{\mathcal{F}} = \mathcal{H}$)*

It is then clear that $\forall\ f \in \mathcal{F}$ the Fréchet gradient $\nabla_{\mathcal{F}} J(f)$, restricted to the subspace $\mathcal{F}$ is equal to the gradient of f in the ambient space $\mathcal{H}$ (since Fréchet gradient is unique). In general, given $\pi \in \Pi$ and $f \in \mathcal{F}$, $\pi + f$ is not necessarily in $\mathcal{F}$. However, the restricted gradient on subspace $\mathcal{F}$ of $J(\pi + f)$ can be defined asymptotically.

Proposition 6.3.8. *Fixing a policy $\pi \in \Pi$, define a sequence of policies $f_k \in \mathcal{F}$, $k = 1, 2, \ldots$ that converges to π: $\lim_{k\to\infty} \|f_k - g\| = 0$, we then have $\lim_{k\to\infty} \|\nabla_{\mathcal{F}} J(f_k) - \nabla_{\mathcal{H}} J(\pi)\|_* = 0$*

Proof. Since Fréchet derivative is a continuous linear operator, we have $\lim_{k\to\infty} \|\nabla_{\mathcal{H}} J(f_k) - \nabla_{\mathcal{H}} J(\pi)\|_* = 0$. By the reasoning above, for $f \in \mathcal{F}$, the gradient $\nabla_{\mathcal{F}} J(f)$ defined via restriction to the space $\mathcal{F}$ does not change compared to $\nabla_{\mathcal{H}} J(f)$, the gradient defined over the ambient space $\mathcal{H}$. Thus we also have $\lim_{k\to\infty} \|\nabla_{\mathcal{F}} J(f_k) - \nabla_{\mathcal{H}} J(\pi)\|_* = 0$. By the same argument, we also have that for any given $\pi \in \Pi$ and $f \in \mathcal{F}$, even if $\pi + f \notin \mathcal{F}$, the gradient $\nabla_{\mathcal{F}} J(\pi + f)$ with respect to the $\mathcal{F}$ can be approximated similarly. $\square$

Note that we are not assuming $J(\pi)$ to be differentiable when restricting to the policy subspace Π.

We now consider the case where Π is not differentiable by parameterization. Note that this does not preclude $J(\pi)$ for $\pi \in \Pi$ to be differentiable in the non-parametric function space. Two complications arise compared to our previous approximate mirror descent procedure. First, for each $\pi \in \Pi$, estimating the gradient $\nabla J(\pi)$ (which may not exist under certain parameterization) can become much more difficult. Second, the update rule $\nabla R(\pi) - \nabla_{\mathcal{F}} J(\pi)$ may not be in the dual space of $\mathcal{F}$, as in the simple case where $\Pi \subset \mathcal{F}$, thus making direct gradient update in the $\mathcal{F}$ space inappropriate.

Assumption 6.3.9. *J is convex in $\mathcal{H}$.*

By convexity of J in $\mathcal{H}$, sub-gradients $\partial J(h)$ exists for all $h \in \mathcal{H}$. In particular, $\partial J(\pi)$ exists for all $\pi \in \Pi$. Note that $\partial J(\pi)$ reflects sub-gradient of π with respect to the ambient policy space $\mathcal{H}$.

We will make use of the following equivalent perspective to mirror descent[13], which consists of two-step process for each iteration t

1. Solve for $h_{t+1} = \operatorname{argmin}_{h \in \mathcal{H}} \eta \langle \partial J(\pi_t), h \rangle + D_R(h, \pi_t)$

2. Solve for $\pi_{t+1} = \operatorname{argmin}_{\pi \in \Pi} D_R(\pi, h_{t+1})$

We will show how this version of the algorithm motivates our main algorithm. Consider step 1 of the main loop of PROPEL, where given a fixed $\pi \in \Pi$, the

optimization problem within $\mathcal{H}$ is

$$(\text{OBJECTIVE_1}) = \min_{h \in \mathcal{H}} \eta \langle \partial J(\pi), h \rangle + D_R(h, \pi) \tag{6.20}$$

Due to convexity of $\mathcal{H}$ and the objective, problem (OBJECTIVE_1) is equivalent to:

$$(\text{OBJECTIVE_1}) = \min \langle \partial J(\pi), h \rangle \tag{6.21}$$

$$\text{s.t. } D_R(h, \pi) \leq \tau \tag{6.22}$$

where τ depends on η. Since π is fixed, this optimization problem can be relaxed by choosing $\lambda \in [0, 1]$, and a set of candidate policies $h = \pi + \lambda f$, for all $f \in \mathcal{F}$, such that $D_R(h, \pi) \leq \tau$ is satisfied (Selection of λ is possible with bounded spaces). Since this constraint set is potentially a restricted set compared to the space of policies satisfying inequality (6.22), the optimization problem (6.20) is relaxed into:

$$(\text{OBJECTIVE_1}) \leq (\text{OBJECTIVE_2}) = \min_{f \in \mathcal{F}} \langle \partial J(\pi), \pi + \lambda f \rangle \tag{6.23}$$

Due to convexity property of J, we have

$$\langle \partial J(\pi), \lambda f \rangle = \langle \partial J(\pi), \pi + \lambda f - \pi) \rangle \leq J(\pi + \lambda f) - J(\pi) \tag{6.24}$$

The original problem OBJECTIVE_1 is thus upper bounded by:

$$\min_{h \in \mathcal{H}} \eta \langle \partial J(\pi), h) \rangle + D_R(h, \pi) \leq \min_{f \in \mathcal{F}} J\big(\pi + \lambda f\big) - J(\pi) + \langle \partial J(\pi), \pi \rangle$$

Thus, a relaxed version of original optimization problem OBJECTIVE_1 can be obtained by miniziming $J(\pi + \lambda f)$ over $f \in \mathcal{F}$ (note that π is fixed). This

naturally motivates using functional regularization technique, such as CORE-RL algorithm [25], to update the parameters of differentiable function f via policy gradient descent update:

$$f' = f - \eta\lambda\nabla_{\mathcal{F}}\lambda J(\pi + \lambda f)$$

where the gradient of J is taken with respect to the parameters of f (neural networks). This is exactly the update step in algorithm 2 (also similar to iterative updte of CORE-RL algorithm), where the neural network policy is regularized by a prior controller π.

Proposition 6.3.10 (Regret bound for the relaxed optimization objective)**.** *Assuming $J(h)$ is L-strongly smooth over $\mathcal{H}$, i.e., $\nabla_{\mathcal{H}} J(h)$ is L-Lipschitz continuous, approximating* $\text{UPDATE}_{\mathcal{H}}$ *by* UPDATE_{F} *per Alg. 2 leads to the expected regret bound:* $\mathbb{E}\left[\frac{1}{T}\sum_{t=1}^{T} J(\pi_t)\right] - J(\pi^*) = O\left(\lambda\sigma\sqrt{\frac{1}{T} + \epsilon} + \lambda^2 L^2\right)$

Proof. Instead of focusing on the bias of the gradient estimate $\nabla_{\mathcal{H}} J(\pi)$, we will shift our focus on the alternative proximal formulation of mirror descent, under optimization and projection errors. In particular, at each iteration t, let $h^*_{t+1} = \text{argmin}_{h\in\mathcal{H}}\, \eta\langle\nabla J(\pi_t), h\rangle + D_R(h, \pi_t)$ and let the optimization error be defined as β_t where $\nabla R(h_{t+1}) = \nabla R(h^*_{t+1}) + \beta_t$. Note here that this is different from (but related to) the notion of bias from gradient estimate of $\nabla J(\pi)$ used in theorem 6.2.1 and theorem 6.2.1. The projection error from imitation learning procedure is defined similarly to theorem 6.2.1: $\pi^*_{t+1} = \text{argmin}_{\pi\in\Pi} D_R(\pi, h_{t+1})$ is the true projection, and $\left\|\pi_{t+1} - \pi^*_{t+1}\right\| \leq \epsilon$.

We start with similar bounding steps to the proof of theorem 6.2.1:

$$\langle \nabla J(\pi_t), \pi_t - \pi \rangle = \frac{1}{\eta} \langle \nabla R(h^*_{t+1}) - \nabla R(\pi_t), \pi_t - \pi \rangle$$
$$= \frac{1}{\eta} \left(\langle \nabla R(h_{t+1}) - \nabla R(\pi_t), \pi_t - \pi \rangle - \langle \beta_t, \pi_t - \pi \rangle \right)$$
$$= \underbrace{\frac{1}{\eta} \left(D_R(\pi, \pi_t) - D_R(\pi, h_{t+1}) + D_R(\pi_t, h_{t+1}) \right)}_{\text{component_1}}$$
$$+ \underbrace{\frac{1}{\eta} \langle \beta_t, \pi_t - \pi \rangle}_{\text{component_2}} \tag{6.25}$$

As seen from the proof of theorem 6.2.1, component_1 can be upperbounded by:

$$\frac{1}{\eta} \Big(\underbrace{D_R(\pi, \pi_t) - D_R(\pi, \pi_{t+1})}_{\text{telescoping}}$$
$$+ \underbrace{D_R(\pi, \pi_{t+1}) - D_R(\pi, \pi^*_{t+1})}_{\text{projection error}}$$
$$\underbrace{-D_R(\pi^*_{t+1}, h_{t+1}) + D_R(\pi_t, h_{t+1})}_{\text{relative improvement}} \Big)$$

The bound on projection error is identical to theorem 6.2.1:

$$D_R(\pi, \pi_t) - D_R(\pi, \pi^*_{t+1}) \leq \epsilon L_R D \tag{6.26}$$

The bound on relative improvement is slightly different:

$$
\begin{aligned}
&D_R(\pi_t, h_{t+1}) - D_R(\pi^*_{t+1}, h_{t+1}) = R(\pi_t) - R(\pi^*_{t+1}) + \langle \nabla R(h_{t+1}), \pi^*_{t+1} - \pi_t \rangle \\
&= R(\pi_t) - R(\pi^*_{t+1} + \langle \nabla R(h^*_{t+1}), \pi^*_{t+1} - \pi_t \rangle) + \langle \beta_t, \pi^*_{t+1} - \pi_t \rangle \\
&\leq \langle \nabla R(\pi_t), \pi_t - \pi^*_{t+1} \rangle - \frac{\alpha}{2} \left\| \pi^*_{t+1} - \pi_t \right\|^2 \\
&\quad + \langle \nabla R(h^*_{t+1}), \pi^*_{t+1} - \pi_t \rangle + \langle \beta_t, \pi^*_{t+1} - \pi_t \rangle \\
&= -\eta \langle \nabla J_{\mathcal{H}}(\pi_t), \pi^*_{t+1} - \pi_t \rangle - \frac{\alpha}{2} \left\| \pi^*_{t+1} - \pi_t \right\|^2 + \langle \beta_t, \pi^*_{t+1} - \pi_t \rangle && (6.27) \\
&\leq \frac{\eta^2}{2\alpha} \left\| \nabla_{\mathcal{H}} J(\pi_t) \right\|_*^2 + \langle \beta_t, \pi^*_{t+1} - \pi_t \rangle \\
&\leq \frac{\eta^2}{2\alpha} L_J^2 + \langle \beta_t, \pi^*_{t+1} - \pi_t \rangle && (6.28)
\end{aligned}
$$

Note here that the gradient $\nabla_{\mathcal{H}} J(\pi_t)$ is not the result of estimation. Combining equations (6.25), (6.26), (6.27), (6.28), we have:

$$
\langle \nabla J(\pi_t), \pi_t - \pi \rangle \leq \frac{1}{\eta} \big(D_R(\pi, \pi_t) - D_R(\pi, \pi_{t+1}) + \epsilon L_R D + \frac{\eta^2}{2\alpha} L_J^2 + \langle \beta_t, \pi^*_{t+1} - \pi \rangle \big) \tag{6.29}
$$

Next, we want to bound β_t. Choose regularizer R to be $\frac{1}{2} \left\| \cdot \right\|^2$ (consistent with the pseudocode in algorithm 2). We have that:

$$
h^*_{t+1} = \operatorname*{argmin}_{h \in \mathcal{H}} \eta \langle \nabla J(\pi_t), h \rangle + \frac{1}{2} \left\| h - \pi_t \right\|^2
$$

which is equivalent to:

$$
h^*_{t+1} = \pi_t + \operatorname*{argmin}_{f \in \mathcal{F}} \eta \langle \nabla J(\pi_t), f \rangle + \frac{1}{2} \left\| f \right\|^2
$$

Let $f^*_{t+1} = \operatorname{argmin}_{f \in \mathcal{F}} \eta \langle \nabla J(\pi_t), f \rangle + \frac{1}{2} \left\| f \right\|^2$. Taking the gradient over f, we can see that $f^*_{t+1} = -\eta \nabla J(\pi_t)$. Let f_{t+1} be the minimizer of $\min_{f \in \mathcal{F}} J(\pi_t + \lambda f)$.

We then have $h^*_{t+1} = \pi_t + f^*_{t+1}$ and $h_{t+1} = \pi + \lambda f_{t+1}$. Thus $\beta_t = h_{t+1} - h^*_{t+1} = f_{t+1} - f^*_{t+1}$.

On one hand, we have

$$J(\pi_t + \lambda f_{t+1}) \leq J(\pi_t + \omega f^*_{t+1}) \leq J(\pi_t) + \langle \nabla J(\pi_t), \omega f^*_{t+1} \rangle + \frac{L}{2} \left\| \omega f^*_{t+1} \right\|^2$$

due to optimality of f_{t+1} and strong smoothness property of J. On the other hand, since J is convex, we also have the first-order condition:

$$J(\pi_t + \lambda f_{t+1}) \geq J(\pi_t) + \langle \nabla J(\pi_t), \lambda f_{t+1} \rangle$$

Combine with the inequality above, and subtract $J(\pi_t)$ from both sides, and using the relationship $f^*_{t+1} = -\eta \nabla J(\pi_t)$, we have that:

$$\langle -\frac{1}{\eta} f^*_{t+1}, \lambda f_{t+1} \rangle \leq \langle -\frac{1}{\eta} f^*_{t+1}, \omega f^*_{t+1} \rangle + \frac{L\omega^2}{2} \left\| f^*_{t+1} \right\|^2$$

Since this is true $\forall \omega$, rearrange and choose ω such that $\frac{\omega}{\eta} - \frac{L\omega^2}{2} = -\frac{\lambda}{2\eta}$, namely $\omega = \frac{1-\sqrt{1-\lambda\eta L}}{L\eta}$, and complete the square, we can establish the bound that:

$$\left\| f_{t+1} - f^*_{t+1} \right\| \leq \eta (\lambda L)^2 B \tag{6.30}$$

for B the upperbound on $\|f_{t+1}\|$. We thus have $\|\beta_t\| = O(\eta(\lambda L)^2)$. Plugging the result from equation 6.30 into RHS of equation 6.29, we have:

$$\langle \nabla J(\pi_t), \pi_t - \pi \rangle \leq \frac{1}{\eta} \big(D_R(\pi, \pi_t) - D_R(\pi, \pi_{t+1}) + \epsilon L_R D + \frac{\eta^2}{2\alpha} L_J^2 \big) + \big(\eta (\lambda L)^2 B \big) \tag{6.31}$$

Since J is convex in $\mathcal{H}$, we have $J(\pi_t) - J(\pi) \leq \langle \nabla J(\pi_t), \pi_t - \pi \rangle$. Similar to theorem 6.2.1, setting $\eta = \sqrt{\frac{1}{\lambda^2 \sigma^2} (\frac{1}{T} + \epsilon)}$ and taking expectation on both sides,

we have:

$$\mathbb{E}\left[\frac{1}{T}\sum_{t=1}^{T} J(\pi_t)\right] - J(\pi^*) = O\big(\lambda\sigma\sqrt{\frac{1}{T}+\epsilon} + \lambda^2 L^2\big) \tag{6.32}$$

Note that unlike regret bound from theorem 6.2.1 under general bias, variance of gradient estimate and projection error, σ^2 here explicitly refers to the bound on neural-network based policy gradient variance. The variance reduction of $\lambda\sigma$, at the expense of some bias, was also similarly noted in a recent functional regularization technique for policy gradient [25]. □

Chapter 7

Experiments and Implementation Details

In this chapter we validate the proposed algorithms empirically, and show that they can discover interpretable, verifiable, generalizable, and performant policies within the PRL framework. The focus of these experiments are on continuous control in simulations. Detailed results are presented of programmatic projections using the NDPS algorithm and of the PROPEL framework with two programmatic classes.

7.1 Environments for Experiments

We generate controllers for cars in *The Open Racing Car Simulator* (TORCS) [96]. TORCS has been used extensively in AI research, for example in [76], [54], and [63] among others. [61] has shown that a Deep Deterministic Policy Gradient (DDPG) network can be used in RL environments with continuous action spaces. The DRL agents for TORCS in this thesis are implementations of the DDPG algorithm, trained on some tracks in the *Practice Mode* of the game.

In its full generality TORCS provides a rich environment with input from up to 89 sensors, and optionally the 3D graphic from a chosen camera

Figure 7.1: Screenshot of a car racing in TORCS. The learning agent has access to the environment state through sensors that provide information about the car and track.

angle in the race. The controllers have to decide the values of 5 parameters during game play, which correspond to the acceleration, brake, clutch, gear and steering of the car. Apart from the immediate challenge of driving the car on the track, controllers also have to make race-level strategy decisions, like making pit-stops for fuel. A lower level of complexity is provided in the *Practice Mode* setting of TORCS. In this mode all race-level strategies are removed. Currently, so far as we know, the state of the art DRL models are capable of racing only in *Practice Mode*, and this is also the environment that we use. Here we consider the input from 29 sensors, and decide values for the acceleration and steering actions.

Due to the sparsity of the lap-time signal, we use a pseudo-reward function during training that provides a heuristic estimate of the agent's performance at each time step. The pseudo-reward used during training is

given by:

$$r_t = V\cos(\theta) - V\sin(\theta) - V|\texttt{trackPos}| \tag{7.1}$$

Here V is the velocity of the car, θ is the angle the car makes with the track axis, and `trackPos` provides the position on the track relative to the track's center. This reward captures the aim of maximizing the longitudinal velocity, minimizing the transverse velocity, and penalizing the agent if it deviates significantly from the center of the track. All the learning algorithms are given access to the same set of sensor readings and rewards from the same pseudo-reward function.

We chose a suite of tracks that provide varying levels of difficulty for the learning algorithms. In particular, for the tracks Ruudskogen and Alpine-2, the DDPG agent is unable to reliably learn a policy that would complete a lap. To evaluate PROPEL we perform the experiments with twenty-five random seeds and report the median lap time over these twenty-five trials. However we note that the TORCS simulator is not deterministic even for a fixed random seed. Since we model the environment as a Markov Decision Process, this non-determinism is consistent with our problem statement.

The sketches used in our experiments provide the basic structure of a proportional-integral-derivative (PID) program, with appropriate holes for parameter and observation values. They have the overall form:

$$K_p(\epsilon - o_i) + K_i \sum_{j=i-N}^{i} (\epsilon - o_j) + K_d(o_{i-1} - o_i) \tag{7.2}$$

Where o_i is the most recent observation provided by the simulator for a chosen sensor, and N is a predetermined constant. We have one controller for each of the actions, acceleration, steering and braking.

These controllers use finite differencing for the derivative term, and the **fold** construct to emulate the integral. To obtain a practical implementation, we constrain the fold calculation to the five latest observations of the history. This constraint corresponds to the standard strategy of automatic (integral) error reset in discretized PID controllers [9].

Each track in TORCS can we viewed as a distinct POMDP. In our implementation of NDPS for TORCS we choose one track and synthesize a program for it. Whenever the algorithm needs to interact with the POMDP, we use the program or DRL agent to race on the track. For example, in the procedure `collect_reward` we use the synthesized program to race one lap, and the reward is a function of the speed, angle and position of the car at each time step.

For the `create_histories` procedure we use the DRL agent to complete one lap of the track. At each time step during the lap, we store the sensor values provided by TORCS along with the action generated by the DRL agent for those values. The `update_histories` procedure uses a two step process. First, the synthesized program is used to race one lap and we store the sensor values provided by TORCS during this lap. Then, we use the DRL agent to generate corresponding actions for the stored sensor values. These pairs, of sensor values and DRL agent actions, are then added to the set of histories.

7.2 Experimental Analysis of NDPS

Now we present an empirical evaluation of the effectiveness of the NDPS algorithm in solving the PRL problem. We synthesize programs for two TORCS tracks, CG-Speedway-1 and Aalborg. These tracks provide varying levels of difficulty, with Aalborg being the more difficult track of the two.

7.2.1 Evaluating Performance

A controller's performance is measured according to two metrics, LAP TIME and REWARD. To calculate the lap time, the programs are allowed to complete a three lap race, and we report the average time taken to complete a lap during this race. The reward function is calculated using the car's velocity, angle with the track axis, and distance from the track axis. The same function is used to train the DRL agent initially. In the experiments we compare the average reward per time step, obtained by the various programs.

We compare among the following RL agents:

A1: DRL. An agent which uses DRL to find a policy represented as a deep neural network. The specific DRL algorithm we use is Deep Deterministic Policy Gradients [60], which has previously been used on TORCS.

A2: *Naive*. Program synthesized without access to a policy oracle.

A3: *NoAug*. Program synthesized without input augmentation.

A4: *NoSketch*. Program synthesized in our policy language without sketch guidance.

A5: *NoIF*. Programs with single PID controllers, without any conditional branching.

A6: NDPS. The Program generated by the NDPS algorithm.

In Table 7.1 we present the performance results of the above list. The lap times in that table are given in minutes and seconds. The TIMEOUT entries indicate that the synthesis process did not return a program that could complete the race, within the specified timeout of twelve hours.

These results justify the various choices that we made in our NDPS algorithm architecture, as discussed in Section 4.2. In many cases those choices were necessary to be able to synthesize a program that could successfully complete a race. As a consequence of these results, we only consider the DRL agent and the NDPS program for subsequent comparisons.

The *NoAug* and *NoSketch* agents are unable to generate programs that complete a single lap on either track. In the case of *NoSketch* this is because the syntax of the policy language (Figure 3.2), defines a very large program space. If we randomly sample from this space without any constraints (like those provided by the sketch), then the probability of getting a good program is extremely low and hence we are unable to reliably generate a program that can complete a lap. The *NoAug* agent performs poorly because without input

Table 7.1: Performance results in TORCS. Lap time is given in Minutes:Seconds. Timeout indicates that the synthesizer did not return a program that completed the race within the specified timeout.

MODEL	CG-SPEEDWAY-1		AALBORG	
	LAP TIME	REWARD	LAP TIME	REWARD
DRL	54.27	118.39	1:49.66	71.23
Naive	2:07.09	58.72	TIMEOUT	–
NoAug	TIMEOUT	–	TIMEOUT	–
NoSketch	TIMEOUT	–	TIMEOUT	–
NoIF	1:01.60	115.25	2:45.13	52.81
NDPS	1:01.56	115.32	2:38.87	54.91

augmentation, the synthesizer is unable to learn the correct behavior once the program deviates even slightly from the oracle's trajectory.

7.2.2 Qualitative Analysis of the Programmatic Policy

We provide qualitative analysis of the inferred programmatic policy through the lens of interpretability, and its behavior in acting in the environment.

Interpretability. Interpretability is a qualitative metric, and cannot be easily demonstrated via experiments. Our agents are interpretable by design, since they are implemented in a human readable policy language. The DRL agents are considered uninterpretable because their policies are encoded in black box neural networks. We can see evidence of this in the compact representation of an acceleration policy found by NDPS, presented in Figure 3.3. The neural policy does not lend itself to such representations.

Table 7.2: Smoothness measure of agents in TORCS, given by the standard deviation of the steering actions during a complete race. Lower values indicate smoother steering.

MODEL	CG-SPEEDWAY-1	AALBORG
DRL	0.5981	0.9008
NDPS	0.1312	0.2483

Behavior of Policy. Our experimental validation showed that the programmatic policy was less aggressive in terms of its use of actions and resulting in smoother steering actions. Numerically, we measure smoothness in Table 7.2 by comparing the population standard deviation of the set of steering actions taken by the program during the entire race. In Figure 7.2 we present a scatter plot of the steering actions taken by the DRL agent and the NDPS program during a slice of the CG-Speedway-1 race. As we can see, the NDPS program takes much more conservative actions.

Robustness to Missing/Noisy Features To evaluate the robustness of the agents with respect to defective sensors we introduce a *Partial Observability* variant of TORCS. In this variant, a random sample of k sensors are declared defective. During the race, one or more of these defective sensors are blocked with some fixed probability. Hence, during gameplay, the sensor either returns the correct reading or a *null* reading. For sufficiently high block probabilities, both agents will fail to complete the race. In Table 7.3 we show the distances raced for two values of the block probability, and in Figure 7.3 we plot the distance raced as we increase the block probability on the Aalborg track. In both

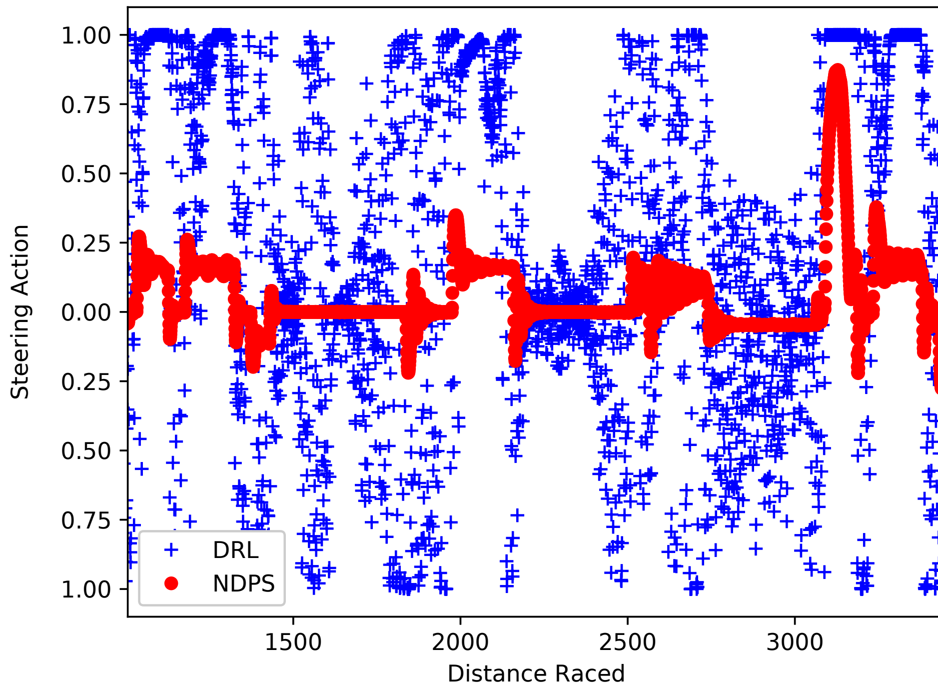

Figure 7.2: Slice of steering actions taken by the DRL and NDPS agents, during the CG-Speedway-1 race. This figure demonstrates that the NDPS agent drives more smoothly.

these experiments, the set of defective sensors was taken to be {`RPM`, `TrackPos`} because we know that the synthesized programs crucially depend on these sensors.

Evaluating Generalization to New Instances To compare the ability of the agents to perform on unseen tracks, we executed the learned policies on tracks of comparable difficulty. For agents trained on the CG-Speedway-1

Table 7.3: Partial observability results in TORCS after blocking sensors {RPM, TrackPos} . For each track and block probability we give the distance, in meters, raced by the program before crashing.

MODEL	CG-SPEEDWAY-1		AALBORG	
	50%	90%	50%	90%
DRL	21	17	71	20
NDPS	1976	200	1477	287

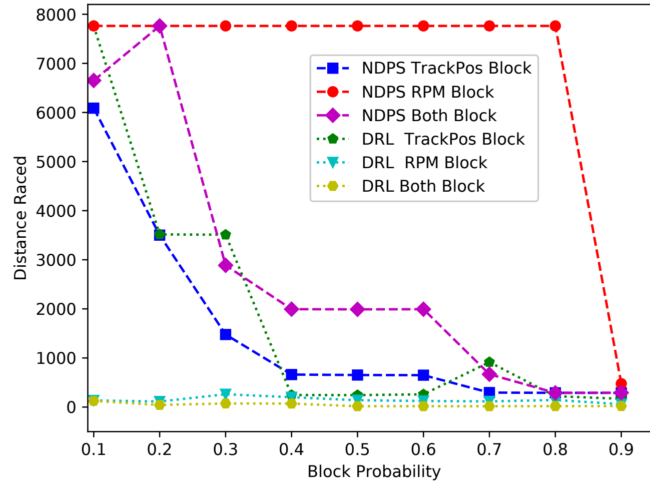

Figure 7.3: Distance raced by the agents as the block probability increases for a particular sensor(s) on Aalborg. The NDPS agent is more robust to blocked sensors.

Table 7.4: Transfer results with training on CG-Speedway-1. 'Cr' indicates that the agent crashed after racing the specified distance.

Model	CG track 2		E-Road	
	Lap Time	Reward	Lap Time	Reward
DRL	Cr 1608m	–	Cr 1902m	–
Ndps	1:40.57	110.18	1:51.59	98.21

Table 7.5: Transfer results with training on Aalborg. 'Cr' denotes the agent crashed, after racing the specified distance.

Model	Alpine 2		Ruudskogen	
	Lap Time	Reward	Lap Time	Reward
DRL	Cr 1688m	–	Cr 3232m	–
Ndps	3:16.68	67.49	3:19.77	57.69

track, we chose CG track 2 and E-Road as the transfer tracks, and for Aalborg trained tracks we chose Alpine 2 and Ruudskogen. As can be seen in Tables 7.4 and 7.5, the Ndps programmatically synthesized program far outperforms the DRL agent on unseen tracks. The DRL agent is unable to complete the race on any of these transfer tracks. This demonstrates the transferability of the policies Ndps finds.

Verifiability of Policies Now we use established symbolic verification techniques to automatically prove two properties of policies generated by Ndps. So far as we know, the current state of the art neural network verifiers cannot verify the DRL network we are using in a reasonable amount of time, due to the size and complexity of the network used to implement the DDPG algorithm.

For example, the Reluplex [49] algorithm was tested on networks at most 300 nodes wide, whereas our network has three layers with 600 nodes each, and other smaller layers.

Smoothness Property For the program given in Figure 3.3 we prove:

$$\forall\ k,\ \sum_{i=k-5}^{k} \|\texttt{RPM}_{i+1} - \texttt{RPM}_i\| < 0.006 \\ \implies \|Acceleration_{k+1} - Acceleration_k\| < 0.49$$

Intuitively, this means that if the sum of the consecutive differences of the last six RPM sensor values is less than 0.006, then the acceleration actions calculated at the last and penultimate step will not differ by more than 0.49. Similarly, for a steering policy, we prove:

$$\forall k\ \sum_{i=k-5}^{k} \|\texttt{TrackPos}_{i+1} - \texttt{TrackPos}_i\| < 0.006 \\ \implies \left\|Steering_{k+1} - Steering_k\right\| < 0.11$$

This proof gives us a guarantee of the type of smooth steering behavior that we empirically examined earlier in this section.

Universal Bounds We can prove that the program in Figure 3.3 satisfies the property:

$$\forall i\ (0 \leq \texttt{RPM}_i \leq 1 \land -1 \leq \texttt{TrackPos}_i \leq 1) \\ \implies (\|Steering_i\| < 101.08 \land -54.53 < Acceleration_i < 53.03).$$

Intuitively, this means that we have proved global bounds for the action values in this environment, assuming reasonable bounds on some of the input values.

In the TORCS environment these bounds are not very useful, since the simulator clips these actions to certain pre-specified ranges. However, this experiment demonstrates that our framework allows us to prove universal bounds on the actions, and this could be a critical property for other environments.

7.2.3 Parameter Optimization

Now we elaborate on the optimization techniques we used in the distance computation step $\text{argmin}_{e'} \sum_{h \in \mathcal{H}} \|e'(h) - e_{NN}(h)\|$, to find a program similar to a given program e, in Algorithm 4. We start by enumerating a list of *program templates*, or programs with numerical-valued parameters θ. This is done by first replacing the numerical constants in e by parameters, eliding some subexpressions from the resulting parameterized program, and then regenerating the subexpressions using the rules of $\mathcal{S}$ (without instantiating the parameters), giving priority to shorter expressions. The resulting program template e_θ follows the sketch $\mathcal{S}$ and is also structurally close to e. Now we search for values for parameters θ that optimally imitate the neural oracle.

Bayesian optimization. We use Bayesian optimization as our primary tool when searching for such optimal parameter values. This method applies to problems in which actions (program outputs) can be represented as vectors of real numbers. All problems considered in our experiments fall in this category. The distance of individual pairs of outputs of the synthesized program and the policy oracle is then simply the Euclidean distance between them. The sum of

these distances is used to define the aggregate cost across all inputs in $\mathcal{H}$. We then use Bayesian optimization to find parameters that minimize this cost.

SMT-based Optimization. We also use a second parameter search technique based on SMT (Satisfiability Modulo Theories) solving. Here, we generate a constraint that stipulates that for each $h \in \mathcal{H}$, the output $e_\theta(h)$ must match $e_{NN}(h)$ up to a constant error. Here, $e_{NN}(h)$ is a constant value obtained by executing e_{NN}. The output $e_\theta(h)$ depends on unknown parameters θ; however, constraints over $e_\theta(h)$ can be represented as constraints over θ using techniques for *symbolic execution* of programs [19]. Because the oracle is only an approximation to the optimal policy in our setting, we do not insist that the generated constraint is satisfied entirely. Instead, we set up a Max-Sat problem which assigns a weight to the constraint for each input h, and then solve this problem with a Max-Sat solver. Unfortunately, SMT-based optimization does not scale well in environments with continuous actions. Consequently, we exclusively use Bayesian optimization for all TORCS based experiments.

The program in Figure 7.4 shows the body of a policy for steering, which together with the acceleration policy given in Figure 3.3, was found by the NDPS algorithm by training on the Aalborg track. Figures 7.5 & 7.6 likewise show the policies for acceleration and steering respectively, when trained on the CG-Speedway-1 track.

$$
\begin{aligned}
& 0.97 * \mathbf{peek}((0.0 - h_{\texttt{TrackPos}}), -1) \\
+ \quad & 0.05 * \mathbf{fold}(+, (0.0 - h_{\texttt{TrackPos}})) \\
+ \quad & 49.98 * (\mathbf{peek}(h_{\texttt{TrackPos}}, -2) - \mathbf{peek}(h_{\texttt{TrackPos}}, -1))
\end{aligned}
$$

Figure 7.4: A programmatic policy for steering, automatically discovered by the NDPS algorithm with training on Aalborg.

$$
\begin{array}{rcl}
\textbf{if} & & (0.0001 - \mathbf{peek}(h_{\texttt{TrackPos}}, -1) > 0) \\
& \textbf{and} & (0.0001 + \mathbf{peek}(h_{\texttt{TrackPos}}, -1) > 0) \\
\textbf{then} & & 0.95 * \mathbf{peek}((0.64 - h_{\texttt{RPM}}), -1) \\
& + & 5.02 * \mathbf{fold}(+, (0.64 - h_{\texttt{RPM}})) \\
& + & 43.89 * (\mathbf{peek}(h_{\texttt{RPM}}, -2) - \mathbf{peek}(h_{\texttt{RPM}}, -1)) \\
\textbf{else} & & 0.95 * \mathbf{peek}((0.60 - h_{\texttt{RPM}}), -1) \\
& + & 5.02 * \mathbf{fold}(+, (0.60 - h_{\texttt{RPM}})) \\
& + & 43.89 * (\mathbf{peek}(h_{\texttt{RPM}}, -2) - \mathbf{peek}(h_{\texttt{RPM}}, -1))
\end{array}
$$

Figure 7.5: A programmatic policy for acceleration, automatically discovered by the NDPS algorithm with training on CG-Speedway-1.

$$
\begin{aligned}
& 0.86 * \mathbf{peek}((0.0 - h_{\texttt{TrackPos}}), -1) \\
+ \quad & 0.09 * \mathbf{fold}(+, (0.0 - h_{\texttt{TrackPos}})) \\
+ \quad & 46.51 * (\mathbf{peek}(h_{\texttt{TrackPos}}, -2) - \mathbf{peek}(h_{\texttt{TrackPos}}, -1))
\end{aligned}
$$

Figure 7.6: A programmatic policy for steering, automatically discovered by the NDPS algorithm with training on CG-Speedway-1.

7.3 Experimental Analysis of PROPEL

We demonstrate the effectiveness of PROPEL in synthesizing programmatic controllers for TORCS. We evaluate over five distinct tracks in the TORCS simulator. The difficulty of a track can be characterized by three properties; track length, track width, and number of turns. Our suite of tracks provides environments with varying levels of difficulty for the learning algorithm. The performance of a policy in the TORCS simulator is measured by the *lap time* achieved on the track. To calculate the lap time, the policies are allowed to complete a three-lap race, and we record the best lap time during this race. We perform the experiments with twenty-five random seeds and report the median lap time over these twenty-five trials. Some of the policies crash the car before completing a lap on certain tracks, even after training for 600 episodes. Such crashes are recorded as a lap time of infinity while calculating the median. If the policy crashes for more than half the seeds, this is reported as CR in Tables 7.1 & 7.8. We choose to report the median because taking the crash timing as infinity, or an arbitrarily large constant, heavily skews other common measures such as the mean.

7.3.1 Evaluating Performance

Baselines. Among recent state-of-the-art approaches to learning programmatic policies are NDPS [95] for high-level language policies, and VIPER [11] for learning tree-based policies. Both NDPS and VIPER rely on imitating a fixed (pre-trained) neural policy oracle, and can be viewed as degenerate

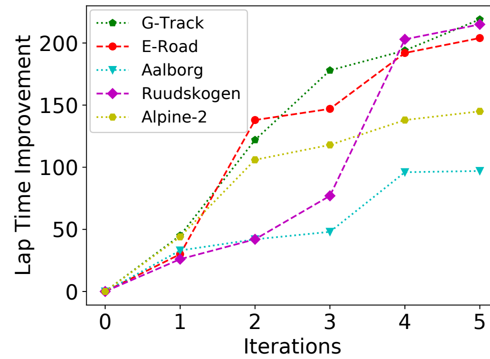

Figure 7.7: Median lap-time improvements during multiple iterations of PropelProg over 25 random seeds.

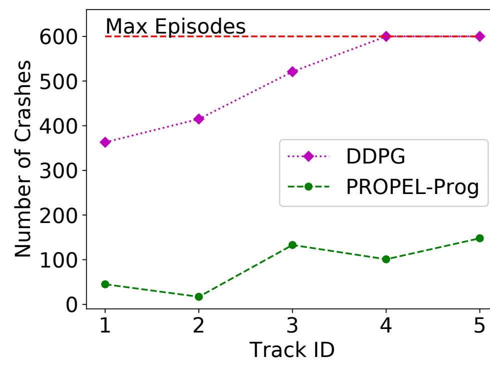

Figure 7.8: Median number of crashes during training of Ddpg and Propel-Prog over 25 random seeds.

Table 7.6: Performance results in TORCS over 25 random seeds. Each entry reports median lap time in seconds over all the seeds (lower is better). A lap time of CR indicates the agent crashed and could not complete a lap for more than half the seeds.

LENGTH	G-TRACK 3186M	E-ROAD 3260M	AALBORG 2588M	RUUDSKOGEN 3274M	ALPINE-2 3774M
PRIOR	312.92	322.59	244.19	340.29	402.89
DDPG	78.82	89.71	101.06	CR	CR
NDPS	108.25	126.80	163.25	CR	CR
VIPER	83.60	87.53	110.57	CR	CR
PROPELPROG	93.67	119.17	147.28	124.58	256.59
PROPELTREE	78.33	79.39	109.83	118.80	236.01

versions of PROPEL that only run Lines 4-6 in Algorithm 1. We present two PROPEL analogues to NDPS and VIPER: (i) PROPELPROG: PROPEL using the high-level language of Figure 3.2 as the class of programmatic policies, similar to NDPS. (ii) PROPELTREE: PROPEL using regression trees, similar to VIPER. We also report results for PRIOR, which is a (sub-optimal) PID controller that is also used as the initial policy in PROPEL. In addition, to study generalization ability as well as safety behavior during training, we also include DDPG, a neural policy learned using the Deep Deterministic Policy Gradients [60] algorithm, with 600 episodes of training for each track. In principle, PROPEL and its analysis can accommodate different policy gradient subroutines. However, in the TORCS domain, other policy gradient algorithms such as PPO and TRPO failed to learn policies that are able to complete the considered tracks. We thus focus on DDPG as our main policy gradient component.

Table 7.7: Performance results in TORCS over 25 random seeds. Each entry reports the ratio of seeds that result in crashes (lower is better).

LENGTH	G-TRACK 3186M	E-ROAD 3260M	AALBORG 2588M	RUUDSKOGEN 3274M	ALPINE-2 3774M
PRIOR	0.0	0.0	0.0	0.0	0.0
DDPG	0.24	0.28	0.40	0.68	0.92
NDPS	0.24	0.28	0.40	0.68	0.92
VIPER	0.24	0.28	0.40	0.68	0.92
PROPELPROG	0.04	0.04	0.12	0.16	0.16
PROPELTREE	0.04	0.04	0.16	0.24	0.36

Tables 7.6& 7.7 show the performance on the considered TORCS tracks. We see that PROPELPROG and PROPELTREE consistently outperform the NDPS [95] and VIPER [11] baselines, respectively. While DDPG outperforms PROPEL on some tracks, its volatility causes it to be unable to learn in some environments, and hence to crash the majority of the time. Figure 7.7 shows the consistent improvements made over the prior by PROPELPROG, over the iterations of the PROPEL algorithm. Figure 7.8 shows that, compared to DDPG, our approach suffers far fewer crashes while training in TORCS.

7.3.2 Variance Reduction

We study the benefits of neurosymbolic policies with respect to variance reduction in detail in [25]. In this section we mention some of those results, and provide a comparison with Target Policy Smoothing Regularization (TPSR) [37]. Many state-of-the-art deep reinforcement learning algorithms use TPSR for variance reduction to achieve reliable learning. In Figure 7.9 we show that

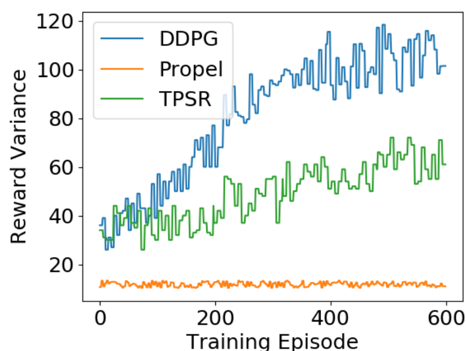

Figure 7.9: Reward variance in TORCS over 25 random seeds for DDPG, PROPEL, and TPSR. The variance is calculated over the performance of all the learning agents for a particular algorithm (each initialized with a different random seed) after each training episode.

neurosymbolic policies learnt via the PROPEL framework have significantly less reward variance than policies learnt via DDPG or TPSR.

In Figure 7.10, we plot laptime improvement over the fixed programmatic policy, so that values above zero denote improved performance. The laps are timed out at 150s, and the objective is to minimize lap-time by completing a lap as fast as possible. Figure 7.10a shows that regularized neurosymbolic controllers perform better on average than the baseline DDPG algorithm, and that we improve upon the programmatic policy with proper regularization. Figure 7.10b shows that intermediate values of λ exhibit good performance, but using the adaptive strategy for setting λ in the TORCS setting gives us the highest-performance policy that significantly beats both the fixed programmatic policy and the DDPG baseline. Also, the variance with the adaptive strategy

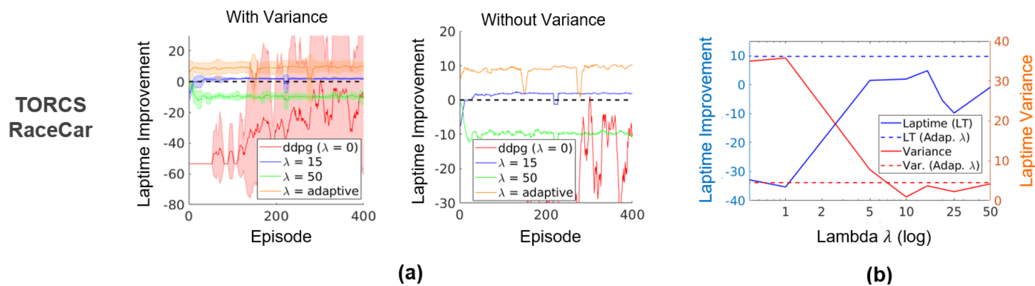

Figure 7.10: Learning results for neurosymbolic policies. (a) Reward improvement over fixed programmatic policy with different set values for λ or an adaptive λ. The right plot is a zoomed-in version of the left plot without variance bars for clarity. (b) Performance and variance in the reward as a function of the regularization λ, across different runs of the algorithm using random initializations/seeds. Dashed lines show the performance (i.e. reward) and variance using the adaptive weighting strategy.

is significantly lower than for the DDPG baseline, which again shows that the learning process *reliably* learns a good controller.

7.3.3 Qualitative Analysis

Evaluating Generalization. To compare the ability of the controllers to perform on tracks not seen during training, we executed the learned policies on all the other tracks (Table 7.8). We observe that DDPG crashes significantly more often than PROPELPROG. This demonstrates the generalizability of the policies returned by PROPEL. Generalization results for the PROPELTREE policy are given in Table 7.9. In general, PROPELTREE policies are more generalizable than DDPG but less than PROPELPROG. On an absolute level, the generalization ability of PROPEL still leaves much room for improvement, which is an interesting direction for future work.

Table 7.8: Generalization results in TORCS, where rows are training and columns are testing tracks. Each entry is formatted as PROPELPROG / DDPG, and the number reported is the median lap time in seconds over all the seeds (lower is better). CR indicates the agent crashed and could not complete a lap for more than half the seeds.

	G-TRACK	E-ROAD	AALBORG	RUUDSKOGEN	ALPINE-2
G-TRACK	-	124 / CR	CR / CR	CR / CR	CR / CR
E-ROAD	102 / 92	-	CR / CR	CR / CR	CR / CR
AALBORG	201 / 91	228 / CR	-	217 / CR	CR / CR
RUUDSKOGEN	131 / CR	135 / CR	CR / CR	-	CR / CR
ALPINE-2	222 / CR	231 / CR	184 / CR	CR / CR	-

Table 7.9: Generalization results in TORCS for PROPELTREE, where rows are training and columns are testing tracks. The number reported is the median lap time in seconds over all the seeds (lower is better). CR indicates the agent crashed and could not complete a lap for more than half the seeds.

	G-TRACK	E-ROAD	AALBORG	RUUDSKOGEN	ALPINE-2
G-TRACK	-	95	CR	CR	CR
E-ROAD	84	-	CR	CR	CR
AALBORG	111	CR	-	CR	CR
RUUDSKOGEN	154	CR	CR	-	CR
ALPINE-2	CR	276	CR	CR	-

In Table 7.9 we show generalization results for the PROPELTREE agent. The generalization results for PROPELTREE are in between those of DDPG and PROPELPROG.

Verifiability of Policies. As shown in prior work [11, 95], parsimonious programmatic policies are more amenable to formal verification than neural policies. Unsurprisingly, the policies generated by PROPELTREE and PROPELPROG are easier to verify than DDPG policies. As a concrete example,

we verified a smoothness property of the PropelProg policy using the Z3 SMT-solver [27]. The verification terminated in 0.49 seconds.

For the program given in Figure 3.3 we proved using symbolic verification techniques, that:

$$\forall k,\ \sum_{i=k}^{k+5} \|\mathbf{peek}(s[\texttt{RPM}], i+1) - \mathbf{peek}(s[\texttt{RPM}], i)\| < 0.003$$
$$\implies \|\mathbf{peek}(a[\texttt{Accel}], k+1) - \mathbf{peek}(a[\texttt{Accel}], k)\| < 0.63$$

Here the function $\mathbf{peek}(., i)$ takes in a history/sequence of sensor or action values and returns the value at position i, . Intuitively, the above logical implication means that if the sum of the consecutive differences of the last six RPM sensor values is less than 0.003, then the acceleration actions calculated at the last and penultimate step will not differ by more than 0.63.

Initialization. In principle, Propel can be initialized with a random program, or a random policy trained using Ddpg. In practice, the performance of Propel depends to a certain degree on the stability of the policy gradient procedure, which is Ddpg in our experiments. Unfortunately, Ddpg often exhibits high variance across trials and fares poorly in challenging RL domains. Specifically, in our Torcs experiments, Ddpg fails on a number of tracks (similar phenomena have been reported in previous work that experiments on similar continuous control domains [41, 25, 95]). Agents obtained by initializing Propel with neural policies obtained via Ddpg also fail on multiple tracks. In contrast, Propel can often finish the challenging tracks when initialized with a very simple hand-crafted programmatic prior.

Table 7.10: Performance results in TORCS of PROPEL agents initialized with neural policies obtained via DDPG, over 25 random seeds. Each entry reports the median lap time in seconds over all the seeds (lower is better). A lap time of CR indicates the agent crashed and could not complete a lap for more than half the seeds.

LENGTH	G-TRACK 3186M	E-ROAD 3260M	AALBORG 2588M	RUUDSKOGEN 3274M	ALPINE-2 3774M
PROPELPROG	97.76	108.06	140.48	CR	CR
PROPELTREE	78.47	85.46	CR	CR	CR

Table 7.11: Performance results in TORCS of PROPEL agents initialized with neural policies obtained via DDPG, over 25 random seeds. Each entry reports the ratio of seeds that result in crashes (lower is better).

LENGTH	G-TRACK 3186M	E-ROAD 3260M	AALBORG 2588M	RUUDSKOGEN 3274M	ALPINE-2 3774M
PROPELPROG	0.12	0.08	0.48	0.68	0.92
PROPELTREE	0.16	0.04	0.56	0.68	0.92

In Tables 7.10 & 7.11 we show the lap time performance and crash ratios of PROPEL agents initialized with neural policies obtained via DDPG. As discussed earlier, DDPG often exhibits high variance across trials and this adversely affects the performance of the PROPEL agents when they are initialized via DDPG.

Chapter 8

Conclusion and Future Work

In this thesis we studied Programmatic Reinforcement Learning (Prl) a framework to leverage partial symbolic knowledge while reliably generating verifiable and performant policies in the Reinforcement Learning (RL) paradigm. Contrasted with the popular Deep Reinforcement Learning (Drl) paradigm, where the policy is represented by a neural network, the aim of these techniques is to generate policies that can be represented in expressive high-level programming languages. Such programmatic policies have several benefits, including being more easily interpreted than neural networks, being amenable to verification by scalable symbolic methods, and having lower variance during learning. The generation methods for programmatic policies also provide a mechanism for systematically using domain knowledge for guiding the search for performant policies. The interpretability and verifiability of these policies provides the opportunity to deploy reinforcement learning based solutions in safety critical environments. This thesis draws on work from both the machine learning and programming languages literature, to create neurosymbolic algorithms that reliably generate performant programmatic policies.

8.1 Neurosymbolic Learning

The two facets of neurosymbolic learning are analogous to the two modes of thought in humans, as detailed by Kahneman's "Thinking, Fast and Slow". Here the neural component is akin to the fast and instinctive "System 1", and symbolic programs embody the more logical and deliberative "System 2". This thesis presents work that establishes that neurosymbolic models provide a principled mechanism to combat the shortcomings of DNN based models, and hence create a promising path towards trustworthy intelligent systems.

In this thesis we formalized the Programmatic Reinforcement Learning (PRL) paradigm, and presented a framework to reliably generate verifiable and performant programmatic policies. By generating complex programs for reinforcement learning environments we have also made advances that are of interest to the programming languages community. The empirical and theoretical analysis of the proposed techniques shows that neurosymbolic learning is a promising research direction and can be used to deploy RL based solutions in challenging environments.

There has been tremendous interest in developing and applying neurosymbolic learning techniques to a variety of domains in recent years. However, it is clear that research in this area has only scratched the surface of the various possibilities, and there remains significant room for research in both effective integration of symbolic techniques in neural architectures, and a systematic exploration of application domains.

8.2 Applications

One avenue of future work is guided by the goal of building neurosymbolic pipelines that can be easily applied to a variety of domains and learning paradigms. Concretely, such pipelines require two distinct components to be user friendly: creating domain specific primitives that have proven efficacy for certain tasks, and algorithmic innovations that are effective at overcoming the challenges of non-differentiable program learning in varied settings. This naturally decomposes the daunting task of creating this neurosymbolic infrastructure, into independent individual task-centric deployments that are promising research topics by themselves. There are a multitude of deployment domains that can be explored with these techniques, and we discuss three broad categories below.

Learning for Control

Significant advances have been made in approaches that perform data-driven control, combining perspectives from machine learning and control theory. By providing a principled mechanism to guide the learner with partial symbolic domain knowledge, this work creates a promising bridge between traditional model-based design and controllers learnt via experience. Control has been the underlying objective in the game simulator we have explored so far. Which has demonstrated the benefits of the provable safety guarantees of programmatic controllers. Deployments of neurosymbolic controllers on cyber-physical systems is also a promising avenue of exploration. Especially since the

robustness introduced by control primitives in the DSL can significantly reduce the performance degradation caused by the Sim2Real gap.

Scientific Discovery

The availability of large amounts of data in almost every scientific field has led to machine learning playing an increasingly important role in scientific discovery. However, their role in devising hypotheses consistent with the data or imagining new experiments is significantly limited by the expressiveness of the models. Neurosymbolic models are capable of overcoming this deficiency. In a recently accepted paper [80], we generate interpretable models for annotating animal behavior datasets. Apart from greatly reducing the burden of manual annotations, these expressive models can be used to guide further experimentation by focusing on factors that are identified by the model as most relevant for desirable behaviors.

Regulated Domains

As AI based systems are included in more user facing applications, regulators are increasingly demanding clearer accountability for decision making processes. Notable examples of such regulatory requirements include the "right to explanation" clause in the European Union's General Data Protection Regulation (GDPR), and "plan of treatment" accountability under medical liability legislation in the US. Consequently,researchers in healthcare and finance need techniques that generate interpretable and certifiable models for use in

consumer facing applications. Concretely, in healthcare Sepsis researchers are working to develop algorithms that generate interpretable adaptive therapies determined by a patient's statistics and their ongoing response to treatment. A key benefit of such a system is that it acts as a recommendation engine for attendant physicians and hence faces fewer regulatory requirements for deployment.

8.3 Theoretical Foundations

Understanding current machine learning approaches from a programming languages perspective is a growing area of research. This area provides a more theoretically grounded approach that uses categories to provide a compositional perspective of learning algorithms. Recent work has shown that gradient descent and backpropagation give a functor from the category of parametrized functions to the category of learning algorithms, and that the category of supervised learning algorithms can be embedded into a certain category of symmetric lenses. This link is particularly interesting since there is substantial literature on the category of symmetric lenses and performing program synthesis for lenses. A rigorous study of these embeddings would help lay the foundations for a more principled understanding of current machine learning techniques, and guide the search for novel neurosymbolic architectures. A close examination of these approaches may also uncover fundamental connections between machine learning and programming languages research.

Creating connections between Machine Learning and other branches of

computer science will help fuel cross-disciplinary collaborations, and generate more reliable real world deployments of intelligent systems. A principled mechanism to integrate symbolic knowledge with gradient based learning provides a promising path to Trustworthy Artificial Intelligence.

Bibliography

[1] Daniel A. Abolafia, Mohammad Norouzi, and Quoc V. Le. Neural program synthesis with priority queue training. *CoRR*, abs/1801.03526, 2018.

[2] Joshua Achiam, David Held, Aviv Tamar, and Pieter Abbeel. Constrained policy optimization. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 22–31. JMLR. org, 2017.

[3] Mohammed Alshiekh, Roderick Bloem, Rudiger Ehlers, Bettina Konighofer, Scott Niekum, and Ufuk Topcu. Safe reinforcement learning via shielding. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[4] Rajeev Alur, Rastislav Bodík, Eric Dallal, Dana Fisman, Pranav Garg, Garvit Juniwal, Hadas Kress-Gazit, P. Madhusudan, Milo M. K. Martin, Mukund Raghothaman, Shambwaditya Saha, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *Dependable Software Systems Engineering*, pages 1–25. IOS Press, 2015.

[5] Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. Scaling enumerative program synthesis via divide and conquer. In *Tools and Algorithms*

for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I, pages 319–336, 2017.

[6] Kiam Heong Ang, Gregory Chong, and Yun Li. Pid control system analysis, design, and technology. *IEEE transactions on control systems technology*, 13(4):559–576, 2005.

[7] Karl J. Åström and Tore Hägglund. *PID controllers: Theory, Design, and Tuning*. Instrument society of America, 1995.

[8] Karl Johan Åström and Tore Hägglund. Automatic tuning of simple regulators with specifications on phase and amplitude margins. *Automatica*, 20(5):645–651, 1984.

[9] K.J. Astrom and T. Hagglund. Automatic tuning of simple regulators with specifications on phase and amplitude margins. *Automatica*, 20(5):645 – 651, 1984.

[10] Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. Deepcoder learning to write programs. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*, 2017.

[11] Osbert Bastani, Yewen Pu, and Armando Solar-Lezama. Verifiable reinforcement learning via policy extraction. In *Advances in Neural*

Information Processing Systems, pages 2494–2504, 2018.

[12] Heinz H Bauschke, Patrick L Combettes, et al. *Convex analysis and monotone operator theory in Hilbert spaces*, volume 408. Springer, 2011.

[13] Amir Beck and Marc Teboulle. Mirror descent and nonlinear projected subgradient methods for convex optimization. *Operations Research Letters*, 31(3):167–175, 2003.

[14] Richard Bellman, Irving Glicksberg, and Oliver Gross. On the "bang-bang" control problem. *Quarterly of Applied Mathematics*, 14(1):11–18, 1956.

[15] Felix Berkenkamp, Matteo Turchetta, Angela Schoellig, and Andreas Krause. Safe model-based reinforcement learning with stability guarantees. In *Advances in neural information processing systems*, pages 908–918, 2017.

[16] Roderick Bloem, Krishnendu Chatterjee, Thomas A. Henzinger, and Barbara Jobstmann. Better quality in synthesis through quantitative objectives. In *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*, pages 140–156, 2009.

[17] Leo Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees*. Wadsworth, 1984.

[18] Rudy Bunel, Matthew Hausknecht, Jacob Devlin, Rishabh Singh, and Pushmeet Kohli. Leveraging grammar and reinforcement learning for neural program synthesis. In *ICLR*, 2018.

[19] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: three decades later. *Communications of the ACM*, 56(2):82–90, 2013.

[20] Kai-Wei Chang, Akshay Krishnamurthy, Alekh Agarwal, Hal Daumé III, and John Langford. Learning to search better than your teacher. In *International Conference on Machine Learning (ICML)*, 2015.

[21] Swarat Chaudhuri, Martin Clochard, and Armando Solar-Lezama. Bridging boolean and quantitative synthesis using smoothed proof search. In *POPL*, pages 207–220, 2014.

[22] Ching-An Cheng, Xinyan Yan, Nathan Ratliff, and Byron Boots. Predictor corrector policy optimization. In *International Conference on Machine Learning (ICML)*, 2019.

[23] Ching-An Cheng, Xinyan Yan, Evangelos Theodorou, and Byron Boots. Accelerating imitation learning with predictive models. In *International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2019.

[24] Ching-An Cheng, Xinyan Yan, Nolan Wagener, and Byron Boots. Fast policy learning through imitation and reinforcement. In *Uncertainty in artificial intelligence*, 2019.

[25] Richard Cheng, Abhinav Verma, Gabor Orosz, Swarat Chaudhuri, Yisong Yue, and Joel Burdick. Control regularization for reduced variance reinforcement learning. In *International Conference on Machine Learning (ICML)*, 2019.

[26] Gal Dalal, Krishnamurthy Dvijotham, Matej Vecerik, Todd Hester, Cosmin Paduraru, and Yuval Tassa. Safe exploration in continuous action spaces. *arXiv preprint arXiv:1801.08757*, 2018.

[27] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *TACAS*, pages 337–340, 2008.

[28] Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdelrahman Mohamed, and Pushmeet Kohli. Robustfill: Neural program learning under noisy i/o. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 990–998. JMLR. org, 2017.

[29] Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdelrahman Mohamed, and Pushmeet Kohli. Robustfill: Neural program learning under noisy I/O. In *ICML*, 2017.

[30] Tao Du, Jeevana Priya Inala, Yewen Pu, Andrew Spielberg, Adriana Schulz, Daniela Rus, Armando Solar-Lezama, and Wojciech Matusik. Inversecsg: automatic conversion of 3d models to CSG trees. *ACM Trans. Graph.*, 37(6):213:1–213:16, 2018.

[31] Yan Duan, Xi Chen, Rein Houthooft, John Schulman, and Pieter Abbeel. Benchmarking deep reinforcement learning for continuous control. In *International Conference on Machine Learning*, pages 1329–1338, 2016.

[32] John C Duchi, Shai Shalev-Shwartz, Yoram Singer, and Ambuj Tewari. Composite objective mirror descent. In *COLT*, pages 14–26, 2010.

[33] Kevin Ellis, Daniel Ritchie, Armando Solar-Lezama, and Josh Tenenbaum. Learning to infer graphics programs from hand-drawn images. In *Advances in Neural Information Processing Systems*, pages 6059–6068, 2018.

[34] John K. Feser, Swarat Chaudhuri, and Isil Dillig. Synthesizing data structure transformations from input-output examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 229–239, 2015.

[35] John K. Feser, Swarat Chaudhuri, and Isil Dillig. Synthesizing data structure transformations from input-output examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 229–239, 2015.

[36] Abraham D Flaxman, Adam Tauman Kalai, and H Brendan McMahan. Online convex optimization in the bandit setting: gradient descent without a gradient. In *Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 385–394. Society for Industrial and Applied Mathematics, 2005.

[37] Scott Fujimoto, Herke van Hoof, and David Meger. Addressing function approximation error in actor-critic methods. In Jennifer G. Dy and

Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, volume 80 of *Proceedings of Machine Learning Research*, pages 1582–1591. PMLR, 2018.

[38] Marta Garnelo, Kai Arulkumaran, and Murray Shanahan. Towards deep symbolic reinforcement learning. *CoRR*, abs/1609.05518, 2016.

[39] Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines. *CoRR*, abs/1410.5401, 2014.

[40] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. Program synthesis. *Foundations and Trends in Programming Languages*, 4(1-2):1–119, 2017.

[41] Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. Deep reinforcement learning that matters. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[42] Jarrett Holtz, Arjun Guha, and Joydeep Biswas. Robot action selection learning via layered dimension informed program synthesis. *CoRR*, abs/2008.04133, 2020.

[43] Radoslav Ivanov, Taylor J. Carpenter, James Weimer, Rajeev Alur, George J. Pappas, and Insup Lee. Case study: verifying the safety of an autonomous racing car with a neural network controller. In Aaron D. Ames, Sanjit A. Seshia, and Jyotirmoy Deshmukh, editors, *HSCC '20:*

23rd ACM International Conference on Hybrid Systems: Computation and Control, Sydney, New South Wales, Australia, April 21-24, 2020, pages 28:1–28:7. ACM, 2020.

[44] Radoslav Ivanov, James Weimer, Rajeev Alur, George J. Pappas, and Insup Lee. Verisig: verifying safety properties of hybrid systems with neural network controllers. In Necmiye Ozay and Pavithra Prabhakar, editors, *Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control, HSCC 2019, Montreal, QC, Canada, April 16-18, 2019*, pages 169–178. ACM, 2019.

[45] Susmit Jha, Sumit Gulwani, Sanjit A Seshia, and Ashish Tiwari. Oracle-guided component-based program synthesis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 215–224. ACM, 2010.

[46] Kishor Jothimurugan, Rajeev Alur, and Osbert Bastani. A composable specification language for reinforcement learning tasks. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 13021–13030, 2019.

[47] Armand Joulin and Tomas Mikolov. Inferring algorithmic patterns with stack-augmented recurrent nets. In *Advances in Neural Information Pro-*

cessing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada, pages 190–198, 2015.

[48] Sham Machandranath Kakade et al. *On the sample complexity of reinforcement learning*. PhD thesis, University of London London, England, 2003.

[49] Guy Katz, Clark W. Barrett, David L. Dill, Kyle Julian, and Mykel J. Kochenderfer. Reluplex: An efficient smt solver for verifying deep neural networks. In *Computer Aided Verification*, pages 97–117, 2017.

[50] Guy Katz, Derek A. Huang, Duligur Ibeling, Kyle Julian, Christopher Lazarus, Rachel Lim, Parth Shah, Shantanu Thakoor, Haoze Wu, Aleksandar Zeljić, David L. Dill, Mykel J. Kochenderfer, and Clark Barrett. The marabou framework for verification and analysis of deep neural networks. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification*, pages 443–452, Cham, 2019. Springer International Publishing.

[51] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[52] W.B̃radley Knox and Peter Stone. Combining manual feedback with subsequent MDP reward signals for reinforcement learning. In *Proc. of 9th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2010)*, May 2010.

[53] Vijay R Konda and John N Tsitsiklis. Actor-critic algorithms. In *Advances in neural information processing systems*, pages 1008–1014, 2000.

[54] Jan Koutník, Giuseppe Cuccu, Jürgen Schmidhuber, and Faustino J. Gomez. Evolving large-scale neural networks for vision-based TORCS. In *Proceedings of the 8th International Conference on the Foundations of Digital Games, FDG 2013, Chania, Crete, Greece, May 14-17, 2013.*, pages 206–212, 2013.

[55] Karol Kurach, Marcin Andrychowicz, and Ilya Sutskever. Neural random-access machines. *CoRR*, abs/1511.06392, 2015.

[56] Brenden M. Lake, Tomer D. Ullman, Joshua B. Tenenbaum, and Samuel J. Gershman. Building machines that learn and think like people. *CoRR*, abs/1604.00289, 2016.

[57] Hoang M. Le, Andrew Kang, Yisong Yue, and Peter Carr. Smooth imitation learning for online sequence prediction. In *International Conference on Machine Learning (ICML)*, 2016.

[58] Hoang M Le, Cameron Voloshin, and Yisong Yue. Batch policy learning under constraints. In *International Conference on Machine Learning (ICML)*, 2019.

[59] Woosuk Lee, Kihong Heo, Rajeev Alur, and Mayur Naik. Accelerating search-based program synthesis using learned probabilistic models. In Jeffrey S. Foster and Dan Grossman, editors, *Proceedings of the 39th ACM*

SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018, pages 436–449. ACM, 2018.

[60] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[61] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *CoRR*, abs/1509.02971, 2015.

[62] Zachary Chase Lipton. The mythos of model interpretability. *CoRR*, abs/1606.03490, 2016.

[63] Daniele Loiacono, Alessandro Prete, Pier Luca Lanzi, and Luigi Cardamone. Learning to overtake in TORCS using simple reinforcement learning. In *Proceedings of the IEEE Congress on Evolutionary Computation, CEC 2010, Barcelona, Spain, 18-23 July 2010*, pages 1–8, 2010.

[64] Sridhar Mahadevan and Bo Liu. Sparse q-learning with mirror descent. In *Proceedings of the Twenty-Eighth Conference on Uncertainty in Artificial Intelligence*, pages 564–573. AUAI Press, 2012.

[65] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K

Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529, 2015.

[66] Grégoire Montavon, Wojciech Samek, and Klaus-Robert Müller. Methods for interpreting and understanding deep neural networks. *CoRR*, abs/1706.07979, 2017.

[67] William H Montgomery and Sergey Levine. Guided policy search via approximate mirror descent. In *Advances in Neural Information Processing Systems*, pages 4008–4016, 2016.

[68] Vijayaraghavan Murali, Swarat Chaudhuri, and Chris Jermaine. Neural sketch learning for conditional program generation. In *ICLR*, 2018.

[69] Arkadii Semenovich Nemirovsky and David Borisovich Yudin. Problem complexity and method efficiency in optimization. *A Wiley-Interscience publication*, 1983.

[70] Emilio Parisotto, Abdel-rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. Neuro-symbolic program synthesis. *arXiv preprint arXiv:1611.01855*, 2016.

[71] Oleksandr Polozov and Sumit Gulwani. Flashmeta: a framework for inductive program synthesis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, pages 107–126, 2015.

[72] Veselin Raychev, Pavol Bielik, Martin T. Vechev, and Andreas Krause. Learning programs from noisy data. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 761–774, 2016.

[73] Stephane Ross and J Andrew Bagnell. Reinforcement and imitation learning via interactive no-regret learning. *arXiv preprint arXiv:1406.5979*, 2014.

[74] Stéphane Ross, Geoffrey Gordon, and Drew Bagnell. A reduction of imitation learning and structured prediction to no-regret online learning. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 627–635, 2011.

[75] Stéphane Ross, Geoffrey J. Gordon, and Drew Bagnell. A reduction of imitation learning and structured prediction to no-regret online learning. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics, AISTATS 2011, Fort Lauderdale, USA, April 11-13, 2011*, pages 627–635, 2011.

[76] Mohammed Salem, Antonio Miguel Mora, Juan J. Merelo, and Pablo García-Sánchez. Driving in TORCS using modular fuzzy controllers. In *Applications of Evolutionary Computation - 20th European Conference, EvoApplications 2017, Amsterdam, The Netherlands, April 19-21, 2017, Proceedings, Part I*, pages 361–376, 2017.

[77] Stefan Schaal. Is imitation learning the route to humanoid robots? *Trends in cognitive sciences*, 3(6):233–242, 1999.

[78] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *International Conference on Machine Learning*, pages 1889–1897, 2015.

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

[80] Ameesh Shah, Eric Zhan, Jennifer J. Sun, Abhinav Verma, Yisong Yue, and Swarat Chaudhuri. Learning differentiable programs with admissible neural heuristics. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020.

[81] Murray Shanahan. Perception as abduction: Turning sensor data into meaningful representation. *Cognitive Science*, 29(1):103–134, 2005.

[82] Xujie Si, Yuan Yang, Hanjun Dai, Mayur Naik, and Le Song. Learning a meta-solver for syntax-guided program synthesis. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*, 2019.

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

[84] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. Mastering chess and Shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*, 2017.

[85] Jasper Snoek, Hugo Larochelle, and Ryan P. Adams. Practical bayesian optimization of machine learning algorithms. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'12, pages 2951–2959, USA, 2012. Curran Associates Inc.

[86] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. Combinatorial sketching for finite programs. In *ASPLOS*, pages 404–415, 2006.

[87] Wen Sun, J Andrew Bagnell, and Byron Boots. Truncated horizon policy search: Combining reinforcement learning & imitation learning. In *International Conference on Learning Representations (ICLR)*, 2018.

[88] Wen Sun, Geoffrey J Gordon, Byron Boots, and J Bagnell. Dual policy iteration. In *Advances in Neural Information Processing Systems*, pages 7059–7069, 2018.

[89] Wen Sun, Arun Venkatraman, Geoffrey J Gordon, Byron Boots, and J Andrew Bagnell. Deeply aggrevated: Differentiable imitation learning for sequential prediction. In *International Conference on Machine Learning (ICML)*, 2017.

[90] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[91] Richard S Sutton, David A McAllester, Satinder P Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Advances in neural information processing systems*, pages 1057–1063, 2000.

[92] Matthew E. Taylor and Peter Stone. Cross-domain transfer for reinforcement learning. In *Proceedings of the Twenty-Fourth International Conference on Machine Learning*, June 2007.

[93] Philip S Thomas, William C Dabney, Stephen Giguere, and Sridhar Mahadevan. Projected natural actor-critic. In *Advances in neural information processing systems*, pages 2337–2345, 2013.

[94] Lazar Valkov, Dipak Chaudhari, Akash Srivastava, Charles Sutton, and Swarat Chaudhuri. Houdini: Lifelong learning as program synthesis. In

Advances in Neural Information Processing Systems, pages 8687–8698, 2018.

[95] Abhinav Verma, Vijayaraghavan Murali, Rishabh Singh, Pushmeet Kohli, and Swarat Chaudhuri. Programmatically interpretable reinforcement learning. In *International Conference on Machine Learning*, pages 5052–5061, 2018.

[96] Bernhard Wymann, Eric Espié, Christophe Guionneau, Christos Dimitrakakis, Rémi Coulom, and Andrew Sumner. TORCS, The Open Racing Car Simulator. http://www.torcs.org, 2014.

[97] Xuesu Xiao, Bo Liu, Garrett Warnell, Jonathan Fink, and Peter Stone. APPLD: Adaptive planner parameter learning from demonstration. *IEEE Robotics and Automation Letters, presented at International Conference on Intelligent Robots and Systems (IROS)*, June 2020.

[98] Tom Zahavy, Nir Ben-Zrihem, and Shie Mannor. Graying the black box: Understanding dqns. *CoRR*, abs/1602.02658, 2016.

[99] He Zhu, Zikang Xiong, Stephen Magill, and Suresh Jagannathan. An inductive synthesis framework for verifiable reinforcement learning. In *ACM Conference on Programming Language Design and Implementation (SIGPLAN)*, 2019.

Vita

Abhinav Verma was born in New Delhi, India. He received the Bachelor of Arts degree in Mathematics from the University of Delhi, and the Master of Science degree in Mathematics from the Indian Institute of Science, Bangalore where he was a recipient of the Junior Research Fellowship awarded by the Council of Scientific & Industrial Research, India. He then enrolled in the University of Oregon from where he received a Master of Science degree in Mathematics, after which he worked at Wolfram Research, Champaign, the makers of Mathematica and Wolfram|Alpha. As a graduate student in the Department of Computer Science at the University of Texas at Austin he has worked under the supervision of Prof. Swarat Chaudhuri at the intersection of Machine Learning and Formal Methods. During his PhD, he received a bronze medal each in the Student Research Competitions at the 45$^{\text{th}}$ ACM SIGPLAN Symposium on Principles of Programming Languages and the 39$^{\text{th}}$ ACM SIGPLAN Conference on Programming Language Design and Implementation, and was awarded the 2020 J.P. Morgan AI Research PhD Fellowship.

Permanent email address: verma@utexas.edu

This dissertation was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.