The Dissertation Committee for Greg Anderson
certifies that this is the approved version of the following dissertation:

Neurosymbolic Approaches to Safe Machine Learning

Committee:

Swarat Chaudhuri, Supervisor

Işıl Dillig, Supervisor

Joydeep Biswas

Rajeev Alur

Neurosymbolic Approaches to Safe Machine Learning

by

Greg Anderson

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

May 2023

Dedicated to my loving partner.

Acknowledgments

First, I want to thank my advisors, Işıl Dillig and Swarat Chaudhuri, for all of their guidance and support over the years. I feel extremely lucky to have had two passionate, dedicated mentors to help me during my Ph.D. They have continually helped me shape and hone my research skills and keep my motivation throughout this process. Outside of research, both Işıl and Swarat have cultivated a friendly, joyful experience, making a difficult journey more pleasant. I couldn't have asked for two better mentors to help me through my graduate studies.

Many times during my doctoral research I was honored to collaborate with excellent scientists and students. I want to recognize Xinyu Wang, Ken McMillan, Shankara Pailoor, Abhinav Verma, and Chenxi Yang for those wonderful collaborative opportunities. Each of them has improved my research skills and impacted my style through our interaction. I want to call out Xinyu in particular for helping me find my feet at the beginning of my Ph.D. and guiding me through my first experiences as a graduate student.

Outside of formal research collaborators, I have also benefitted immensely from all of my friends in the UToPiA and Trishul research groups. The community of these groups, both inside and outside the office was critical to my success and to my happiness during my time as a graduate student. I

want to thank Kostas Ferles, Yuepeng Wang, Jia Chen, Yu Feng, Navid Yaghmazadeh, Jiayi Wei, Jon Stephens, Jocelyn Chen, Ben Mariano, Ben Sepanski, Ziteng Wang, Celeste Barnaby, Ian Kretz, Noah Patton, Anders Miltner, Calvin Smith, Dipak Chaudhari, Sam Anklesaria, Josh Hoffman, Atharva Sehgal, Meghana Sistla, Yeming Wen, Dweep Trivedi, Eric Hsiung, Amitayush Thakur, Thomas Logan, and Christopher Hahn.

I am indebted to the members of my thesis committee, Joydeep Biswas and Rajeev Alur. Their feedback has been critical in shaping this dissertation and my work going forward into the future.

None of this would have happened without the people who first introduced me to research. I want to express my appreciation for John Knight, my first research mentor, along with César Muñoz and Aaron Dutle for introducing me to formal methods and program analysis. I credit these three, and my time at the University of Virginia and NASA, with turning me toward a career in research and encouraging me to pursue a Ph.D. in the first place.

Finally, I want to thank all of the people outside the research community who have kept me going for the past six years. Foremost among these is my partner, who provided boundless support over the course of my Ph.D., and without whom this dissertation would never have been written. I would also like to thank my parents for setting me up for success in this endeavor and supporting me throughout. Lastly, the rest of my family and my friends in Austin deserve thanks for keeping me connected to the world outside of academia and for their support.

Neurosymbolic Approaches to Safe Machine Learning

Publication No. _____

Greg Anderson, Ph.D.
The University of Texas at Austin, 2023

Supervisors: Swarat Chaudhuri
Işıl Dillig

Neural networks have shown immense promise in solving a variety of challenging problems including computer vision, security, and robotic control. However these applications often come with substantial risk, and in order to deploy machine learning systems in the real world, we need tools to analyze the behavior of these systems. This presents a problem to researchers because neural networks are generally resistant to traditional approaches to program analysis. From a formal analysis perspective, networks are high-dimensional and existing tools simply cannot scale enough to handle them. From a testing perspective, networks are known to be subject to "adversarial examples", which are specific, sparse inputs that trigger unsafe behavior. In this work, we explore two different approaches to analyze systems with neural network components.

First, we consider the problem of analyzing neural networks directly. In this portion of the work, we develop an efficient approach to verify the

robustness of neural networks. In order to do this, we use machine learning techniques to develop heuristics which drastically improve the efficiency of existing program analysis approaches to robustness analysis. We show that this synergystic combination of machine learning and symbolic analysis is able to outperform existing approaches to robustness verification across a large suite of benchmarks.

Second, we develop techniques for bypassing the analysis of neural networks entirely, instead relying on external structures to enforce safety. The core idea here is to develop the network together with a *shield*, a traditional program which is attempting to achieve the same goal as the network. The shield is unlikely to reach the same level of performance as a neural network, but is more amenable to verification. By carefully combining the network and the shield, we maintain the safety of the shield while incorporating the performance of the neural network. We explore different variations on this idea in different contexts, and show that we are able to achieve safe policies while maintaining most of the performance benefits of neural networks.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Deep neural networks (DNN's) have show enormous popularity for a wide spectrum of applications, ranging from image recognition [45,55] and malware detection [106, 107] to machine translation [101]. Due to their effectiveness in practice, deep learning has also found numerous uses in safety-critical systems, including self-driving cars [10, 16], unmanned aerial systems [50], and medical diagnosis [26]. In all of these applications, machine learning allows us to push past the boundaries of existing algorithmic techniques and solve previously intractable problems.

At the same time, neural networks come with a significant downside compared to traditional programs: they are extremely difficult to interpret or analyze. Both during and after training, the behavior of neural networks is highly non-smooth, and susceptible to pathological changes when given seemingly random inputs [27, 76]. Because of this, it is difficult to deploy machine learning in real-world applications where we must trust the system to behave in some "reasonable" way. In order to make such deployment attractive, new techniques are needed to analyze and constrain machine learning systems, to ensure that their behavior is acceptable when they are used in contexts with

substantial consequences for failure.

In this dissertation, we will consider two different problems in machine learning safety. The first is to show when neural networks are *robust*, i.e., their behavior is not drastically changed by small changes in the input. The second problem concerns verifying the safety of reinforcement learning systems, which often need to operate in real-world environments.

1.1 Robustness

Despite their widespread use in a broad range of application domains, it is well-known that deep neural networks are vulnerable to *adversarial counterexamples*, which are small perturbations to a network's input that cause the network to output incorrect labels [27, 76]. For instance, Figure 1.1 shows two adversarial examples in the context of speech recognition and image classification. As shown in the top half of Figure 1.1, two sound waves that are virtually indistinguishable are recognized as "How are you?" and "Open the door" by a DNN-based speech recognition system [38]. Similarly, as illustrated in the bottom half of the same figure, applying a tiny perturbation to a panda image causes a DNN to misclassify the image as that of a gibbon.

It is by now well-understood that such adversarial counterexamples can pose serious security risks [105]. Prior work [12,35,51,77,78] has advocated the property of *local robustness* (or *robustness* for short) for protecting neural networks against attacks that exploit such adversarial examples. To understand what robustness means, consider a neural network that classifies an input $\boldsymbol{x}$

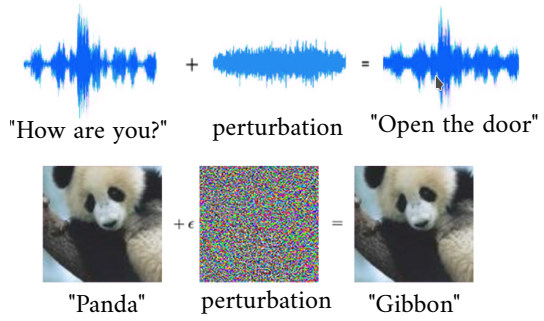


Figure 1.1: Small perturbations of the input cause the sound wave and the image to be misclassified.

as having label $\boldsymbol{y}$. Then, local robustness requires that all inputs $\boldsymbol{x}'$ that are "very similar" [1] to $\boldsymbol{x}$ are also classified as having the same label $\boldsymbol{y}$.

Due to the growing consensus on the desirability of robust neural networks, many recent efforts have sought to algorithmically analyze robustness of networks. Of these, one category of methods seeks to discover adversarial counterexamples using numerical optimization techniques such as Projected Gradient Descent (PGD) [68] and Fast Gradient Sign Method (FGSM) [39]. A second category aims to *prove* network robustness using symbolic methods ranging from SMT-solving [51] to abstract interpretation [35, 100]. These two categories of methods have complementary advantages. Numerical counterexample search methods can quickly find violations, but are "unsound", in that they fail to offer certainty about a network's robustness. In contrast, proof search methods are sound, but they are either incomplete [35] (i.e., suffer from false positives) or do not scale well [51].

[1] For example, "very similar" may mean $\boldsymbol{x}'$ is within some ϵ distance from $\boldsymbol{x}$, where distance can be measured using different metrics such as L^2 norm.

In Chapter 2, we will look at a novel algorithm for analyzing the robustness of neural networks which combines proof-based and optimization-based approaches to analyzing neural network robustness. This combination allows our analysis to be faster and/or more precise than other proof-based approaches to robustness analysis, and more precise and reliable than optimization-based approaches. Moreover, the technique we develop in Chapter 2 is able to improve its performance by using another set of machine learning algorithms to analyze the properties it is trying to prove and optimize its proof strategy accordingly.

1.2 Safe Reinforcement Learning

Aside from robustness, safety is also important in the context of deep reinforcement learning (RL), in which a neural network is used to control an agent interacting with the real world. Ensuring safety in this setting is a fundamental challenge which must be solved in order to deploy RL in real-world, safety-critical systems [3, 33]. Additionally, RL introduces a new set of challenges for verification compared to the supervised learning setting considered in Chapter 2. First, because an RL agent interacts with an environment over time, we must consider the *long-term effects* of network behavior. Second, because RL agents are usually trained by using a neural network to gather data from the environment, we must not only ensure the safety of *one* network, but rather *every* network which is used to gather data.

Existing work in safe RL can be categorized along two axes, addressing

different situations that arise from real-world problems. First, we can consider what kinds of safety guarantees are offered. Some techniques show that a system is safe *with high probability* [3, 15, 20, 73] and others give *deterministic* guarantees which ensure that the system is safe *in all* cases [11, 13, 109]. Second, different techniques offer these safety guarantees at different times. Some tools only ensure that the final learned system is safe [109] while others ensure that a system does not behave unsafely even during training time [5, 20].

In this dissertation, we will consider techniques which aim to achieve safety during training as well as at convergence. In order to do this efficiently and precisely, we combine existing RL algorithms with novel ideas in program analysis and synthesis to develop *neurosymbolic* learning algorithms. These algorithms are designed to combine the best aspects of machine learning with the best aspects of traditional programs in order to achieve both high performance and safety. Specifically, our neurosymbolic algorithms combine a neural agent with a *shield* which can monitor the actions of the agent and intervene when those actions may lead to unsafe behavior.

Safety in Known Environments In Chapter 4, we will consider situations in which the behavior of the environment can be approximated beforehand. This kind of scenario can arise in a variety of robotics applications where it is possible to develop a nominal environment model before collecting any data. In this case, we use ideas from formal verification research to synthesize shields which are proven to be *deterministically* safe. Compared to prior work which

used *statistical* approaches to measure safety, our neurosymbolic approach is much more effective at preventing potentially dangerous behavior. Moreover, by iteratively improving the shield over time, we are able to achieve better performance than prior shielding approaches based on formal methods. This combination of machine learning algorithms with ideas from formal analysis and program synthesis allows us to learn policies which are both safe and highly effective.

Safety in Unkonwn Environments In Chapter 5, we will consider cases where the behavior of the environment is *a priori* unknown. We do this by learning a model of the environment as we go, and applying formal methods algorithms to the learned model. Compared to Chapter 4 this allows us to consider a much broader range of environments. As a tradeoff, the technique developed in Chapter 5 is not able to *guarantee* safety. Instead it ensure safe behavior with high probability. Even so, we find that this approach achieves better empirical safety than prior work based on neural models of safe behavior while maintaining comparable performance.

Scalable Safe Exploration The techniques presented in Chapters 4 and 5 both suffer from scalability challenges which make it difficult to apply them to complex environments. In Chapter 6, we will look at an alternative approach to safe exploration which addresses these challenges. This approach leaves behind some of the more structured neurosymbolic ideas of the previous chapters, instead relying on neural networks and gradient-based learning to ensure

safety. This once again weakens the safety guarantees we are able to provide, but drastically increases the scope of the technique. Chapters 4, 5, and 6 together form a spectrum of safe exploration approaches which are applicable to a broad class of systems with different complexities and different levels of *a priori* knowledge.

Chapter 2

Robustness Verification[1]

First, we will consider the problem of robustness analysis: how do we ensure that the behavior of a neural network is stable under small changes in the input? In this chapter, we present a new technique for robustness analysis of neural networks that combines the best of proof-based and optimization-based methods. Our approach combines formal reasoning techniques based on *abstract interpretation* with continuous and black-box optimization techniques from the machine learning community. This tight coupling of optimization and abstraction has two key advantages: First, optimization-based methods can efficiently search for counterexamples that prove the violation of the robustness property, allowing efficient falsification in addition to verification. Second, optimization-based methods provide a data-driven way to automatically refine the abstraction when the property can be neither falsified nor proven.

The workflow of our approach is shown schematically in Figure 2.1 and consists of both a *training* and a *deployment* phase. During the training phase, our method uses black-box optimization techniques to learn a so-called *verification policy* π_θ from a representative set of training problems. Then the

[1]This chapter is based on [7]. The author of this dissertation was responsible for the main ideas along with the theoretical analysis and most of the implementation.

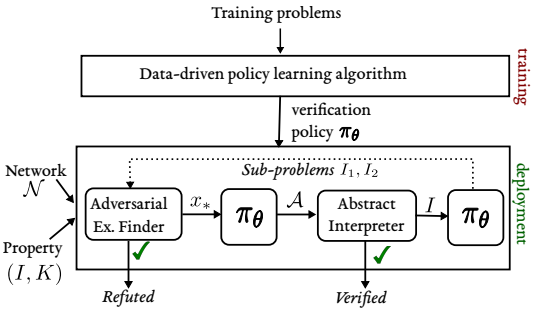


Figure 2.1: Schematic overview of our approach.

deployment phase uses the learned verification policy to guide how gradient-based counterexample search should be coupled with proof synthesis for solving previously-unseen verification problems.

The input to the deployment phase of our algorithm consists of a neural network $\mathcal{N}$ as well as a robustness specification (I, K) which states that all points in the *input region* I should be classified as having label K. Given this input, our algorithm first uses gradient-based optimization to search for an adversarial counterexample, which is a point in the input region I that is classified as having label $K' \neq K$. If we can find such a counterexample, then the algorithm terminates with a witness to the violation of the property. However, even if the optimization procedure fails to find a true counterexample, the result $\boldsymbol{x}_*$ of the optimization problem can still convey useful information. In particular, our method uses the learned verification policy π_θ to map all available information, including $\boldsymbol{x}_*$, to a promising abstract domain $\mathcal{A}$ to use when attempting to verify the property. If the property can be verified using domain $\mathcal{A}$, then the algorithm successfully terminates with a robustness proof.

In cases where the property is neither verified nor refuted, our algorithm uses the verification policy π_θ to *split* the input region I into two sub-regions I_1, I_2 such that $I = I_1 \cup I_2$ and tries to verify/falsify the robustness of each region separately. This form of refinement is useful for both the abstract interpreter as well as the counterexample finder. In particular, since gradient-based optimization methods are not guaranteed to find a global optimum, splitting the input region into smaller parts makes it more likely that the optimizer can find an adversarial counterexample. Splitting the input region is similarly useful for the abstract interpreter because the amount of imprecision introduced by the abstraction is correlated with the size of the input region.

As illustrated by the discussion above, a key part of our verification algorithm is the use of a policy π_θ to decide (a) which abstract domain to use for verification, and (b) how to split the input region into two sub-regions. Since there is no obvious choice for either the abstract domain or the splitting strategy, our algorithm takes a *data-driven approach* to learn a suitable verification policy π_θ during a training phase. During this training phase, we use a black-box optimization technique known as Bayesian optimization to learn values of θ that lead to strong performance on a representative set of verification problems. Once this phase is over, the algorithm can be deployed on networks and properties that have not been encountered during training.

Our proposed verification algorithm has some appealing theoretical properties in that it is both sound and δ-complete [32]. That is, if our method verifies the property (I, K) for network $\mathcal{N}$, this means that $\mathcal{N}$ does indeed

classify all points in the input region I as belonging to class K. Furthermore, our method is δ-complete in the sense that, if the property is falsified with counterexample $\boldsymbol{x}_*$, this means that $\boldsymbol{x}_*$ is within δ of being a true counterexample.

We have implemented the proposed method in a tool called CHARON [2], and used it to analyze hundreds of robustness properties of ReLU neural networks, including both fully-connected and convolutional neural networks, trained on the MNIST [59] and CIFAR [54] datasets. We have also compared our method against state-of-the-art network verification tools (namely, RELUPLEX, RELUVAL, and AI^2) and shown that our method outperforms all prior verification techniques, either in terms of accuracy or performance or both. In addition, our experimental results reveal the benefits of learning to couple proof search and optimization.

In all, this chapter makes the following key contributions:

- We present a new sound and δ-complete decision procedure that combines abstract interpretation and gradient-based counterexample search to prove robustness of deep neural networks.
- We describe a method for automatically learning verification policies that direct counterexample search and abstract interpretation steps carried out during the analysis.

[2] Complete Hybrid Abstraction Refinement and Optimization for Neural Networks.

- We conduct an extensive experimental evaluation on hundreds of benchmarks and show that our method significantly outperforms state-of-the-art tools for verifying neural networks. For example, our method solves 2.6× and 16.6× more benchmarks compared to ReluVal and Reluplex respectively.

2.1 Background

In this section, we provide some background on neural networks and robustness.

2.1.1 Neural Networks

For the purposes of this chapter, we will define a neural network as a function $\mathcal{N} : \mathbb{R}^n \to \mathbb{R}^m$ of the form $L_1 \circ \sigma_1 \circ \cdots \circ \sigma_{k-1} \circ L_k$, where each L_i is a differentiable *layer* and each σ_i is a non-linear, almost-everywhere differentiable *activation function*. While there are many types of activation functions, the most popular choice in modern neural networks is the *rectified linear unit (ReLU)*, defined as $\text{ReLU}(x) = \max(x, 0)$. This function is applied element-wise to the output of each layer except the last. In this work, we consider feed-forward and convolutional networks, which have the additional property of being Lipschitz-continuous.

We think of each layer L_i as an affine transformation $(\boldsymbol{W}, \boldsymbol{b})$ where $\boldsymbol{W}$ is a *weight matrix* and $\boldsymbol{b}$ is a *bias vector*. Thus, the output of the i'th layer is computed as $\boldsymbol{y} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$. We note that both *fully-connected* as well as

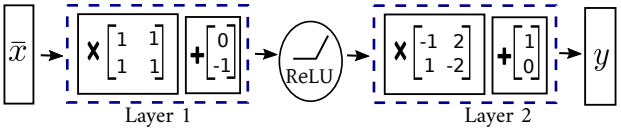


Figure 2.2: A feedforward network implementing XOR

convolutional layers can be expressed as affine transformations [35]. While our approach can also handle other types of layers (e.g., max pooling), we only focus on affine transformations to simplify presentation.

In this chapter, we consider networks used for classification tasks. That is, given some input $\boldsymbol{x} \in \mathbb{R}^n$, we wish to put $\boldsymbol{x}$ into one of m *classes*. The output of the network $\boldsymbol{y} \in \mathbb{R}^m$ is interpreted as a vector of *scores*, one for each class. Then $\boldsymbol{x}$ is put into the class with the highest score. More formally, given some input $\boldsymbol{x}$, we say the network $\mathcal{N}$ assigns $\boldsymbol{x}$ to a class K if $(\mathcal{N}(\boldsymbol{x}))_K > (\mathcal{N}(\boldsymbol{x}))_j$ for all $j \neq K$.

Example Figure 2.2 shows a 2-layer feedforward neural network implementing the XOR function. To see why this network "implements" XOR, consider the vector $[0\ 0]^\top$. After applying the affine transformation from the first layer, we obtain $[0\ -1]^\top$. After applying ReLU, we get $[0\ 0]^\top$. Finally, after applying the affine transform in the second layer, we get $[1\ 0]^\top$. Because the output at index zero is greater than the output at index one, the network will classify $[0\ 0]^\top$ as a zero. Similarly, this network classifies both $[0\ 1]^\top$ and $[1\ 0]^\top$ as 1 and $[1\ 1]^\top$ as 0.

2.1.2 Robustness

(Local) robustness [12] is a key correctness property of neural networks which requires that all inputs within some region of the input space fall within the same region of the output space. Since we focus on networks designed for classification tasks, we will define "the same region of the output space" to mean the region which assigns the same class to the input. That is, a robustness property asserts that a small change in the input cannot change the class assigned to that input.

More formally, a *robustness property* is a pair (I, K) with $I \subseteq \mathbb{R}^n$ and $0 \leq K \leq m-1$. Here, I defines some region of the input that we are interested in and K is the class into which all the inputs in I should be placed. A network $\mathcal{N}$ is said to satisfy a robustness property (I, K) if for all $\boldsymbol{x} \in I$, we have $(\mathcal{N}(\boldsymbol{x}))_K > (\mathcal{N}(\boldsymbol{x}))_j$ for all $j \neq K$.

Example Consider the following network with two layers:

$$\mathcal{N}(\boldsymbol{x}) = \begin{bmatrix} 2 & 1 \\ -1 & 1 \end{bmatrix} \text{ReLU}\left(\begin{bmatrix} 1 \\ 2 \end{bmatrix} \boldsymbol{x} + \begin{bmatrix} -1 \\ 1 \end{bmatrix} \right) + \begin{bmatrix} 1 \\ 2 \end{bmatrix}.$$

For the input $\boldsymbol{x} = 0$, we have $\mathcal{N}(0) = [2\ 3]^\top$; thus, the network outputs label 1 for input 0. Let $I = [-1, 1]$ and $K = 1$. Then for all $\boldsymbol{x} \in I$, the output of $\mathcal{N}$ is of the form $[a+1\ a+2]^\top$ for some $a \in [0, 3]$. Therefore, the network classifies every point in I as belonging to class 1, meaning that the network is robust in $[-1, 1]$. On the other hand, suppose we extend this interval to $I' = [-1, 2]$. Then $\mathcal{N}(2) = [8\ 6]^\top$, so $\mathcal{N}$ assigns input 2 as belonging to class 0. Therefore $\mathcal{N}$ is *not* robust in the input region $[-1, 2]$.

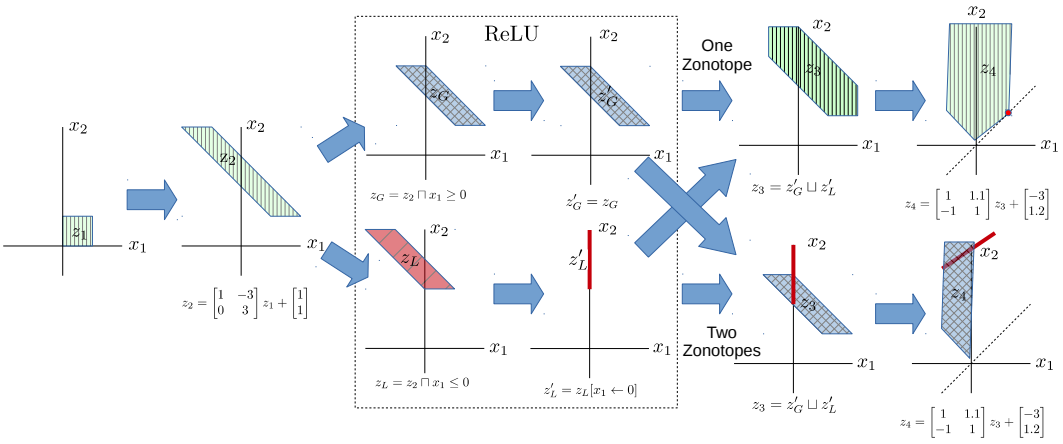


Figure 2.3: Zonotope analysis of a neural network.

2.1.3 Abstract Interpretation for Neural Networks

In this chapter, we build on the prior AI^2 work [35] for analyzing neural networks using the framework of *abstract interpretation* [22]. AI^2 allows analyzing neural networks using a variety of numeric abstract domains, including intervals (boxes) [22], polyhedra [90], and zonotopes [36]. In addition, AI^2 also supports *bounded powerset domains* [22], which essentially allow a bounded number of disjunctions in the abstraction. Since the user can specify any number of disjunctions, there are *many* different abstract domains to choose from, and the precision and scalability of the analysis crucially depend on one's choice of the abstract domain.

The following example illustrates a robustness property that can be verified using the bounded powersets of zonotopes domain with two disjuncts but not with intervals or plain zonotopes:

Example Consider a network defined as:

$$\mathcal{N}(\boldsymbol{x}) = \begin{bmatrix} 1 & 1.1 \\ -1 & 1 \end{bmatrix} \text{ReLU}\left(\begin{bmatrix} 1 & -3 \\ 0 & 3 \end{bmatrix} \boldsymbol{x} + \begin{bmatrix} 1 \\ 1 \end{bmatrix} \right) + \begin{bmatrix} -3 \\ 1.2 \end{bmatrix}$$

Now, suppose we want to verify that for all $\boldsymbol{x} \in [0, 1]^2$, the network assigns class 1 to $\boldsymbol{x}$.

Let us now analyze this network using the zonotope abstract domain, which overapproximates a given region using a zonotope (i.e., center-symmetric polytope). The analysis of this network using the zonotope domain is illustrated in Figure 2.3. At first, the initial region is propagated through the affine transformation as a single zonotope. Then, this zonotope is split into two pieces, the blue (crosshatched) one for which $\boldsymbol{x}_1 \geq 0$ and the red (diagonally striped) one for which $\boldsymbol{x}_1 \leq 0$. The ReLU transforms the red piece into a line. (We omit the ReLU over $\boldsymbol{x}_2$ because it does not change the zonotopes in this case.) After the ReLU, we show two cases: on top is the plain zonotope domain, and on the bottom is a powerset of zonotopes domain. In the plain zonotope domain, the abstraction after the ReLU is the join of the blue and red zonotopes, while in the powerset domain we keep the blue and red zonotopes separate. The final images show how the second affine transformation affects all three zonotopes.

This example illustrates that the propery cannot be verified using the plain zonotope domain, but it *can* be verified using the powerset of zonotopes domain. Specifically, observe that the green (vertically striped) zonotope at the top includes the point $[1.2\ 1.2]^\top$ (marked by a dot), where the robustness

specification is violated. On the other hand, the blue and red zonotopes obtained using the powerset domain do not contain any unsafe points, so the property is verified using this more precise abstraction.

2.2 Algorithm for Checking Robustness

In this section, we describe our algorithm for checking robustness properties of neural networks. Our algorithm interleaves optimization-based counterexample search with proof synthesis using abstraction refinement. At a high level, abstract interpretation provides an efficient way to verify properties but is subject to false positives. Conversely, optimization based techniques for finding counterexamples are efficient for finding adversarial inputs, but suffer from false negatives. Our algorithm combines the strengths of these two techniques by searching for both proofs and counterexamples at the same time and using information from the counterexample search to guide proof search.

Before we describe our algorithm in detail, we need to define our optimization problem more formally. Given a network $\mathcal{N}$ and a robustness property (I, K), we can view the search for an adversarial counterexample as the following optimization problem:

$$\boldsymbol{x}_* = \arg\min_{\boldsymbol{x} \in I} \left(\mathcal{F}(\boldsymbol{x})\right) \tag{2.1}$$

where our *objective function* $\mathcal{F}$ is defined as follows:

$$\mathcal{F}(\boldsymbol{x}) = \left(\mathcal{N}(\boldsymbol{x})\right)_K - \max_{j \neq K} \left(\mathcal{N}(\boldsymbol{x})\right)_j \tag{2.2}$$

Intuitively, the objective function $\mathcal{F}$ measures the difference between the score for class K and the maximum score among classes other than K. Note that if the value of this objective function is not positive at some point $\boldsymbol{x}$, then there exists some class which has a greater (or equal) score than the target class, so point $\boldsymbol{x}$ constitutes a true adversarial counterexample.

The optimization problem from Equation 2.1 is clearly useful for searching for counterexamples to the robustness property. However, even if the solution $\boldsymbol{x}_*$ is not a true counterexample (i.e., $\mathcal{F}(\boldsymbol{x}_*) > 0$), we can still use the result of the optimization problem to guide proof search.

Based on this intuition, we now explain our decision procedure, shown in Algorithm 2.1, in more detail. The VERIFY procedure takes as input a network $\mathcal{N}$, a robustness property (I, K) to be verified, and a so-called *verification policy* π_θ. The verification policy is used to decide what kind of abstraction to use and how to split the input region when attempting to verify the property. In more detail, the verification policy π_θ, parameterized by θ, is a pair $(\pi^\alpha_\theta, \pi^I_\theta)$, where π^α_θ is a (parameterized) function known as the *domain policy* and π^I_θ is a function known as the *partition policy*. The domain policy is used to decide which abstract domain to use, while the partition policy determines how to split the input region I into two partitions to be analyzed separately. In general, it is quite difficult to write a good verification policy by hand because there are many different parameters to tune and neural networks are quite opaque and difficult to interpret. In Section 2.3, we explain how the parameters of these policy functions are learned from data.

Algorithm 2.1 The main algorithm

procedure VERIFY($\mathcal{N}, I, K, \pi_\theta$)
  Input: Network N, robustness property (I, K), policy $\pi_\theta = (\pi_\theta^\alpha, \pi_\theta^I)$
  Output: Counterexample if $\mathcal{N}$ is not robust, or Verified.
  $\boldsymbol{x}_* \leftarrow$ MINIMIZE$(I, \mathcal{F})$
  if $\mathcal{F}(\boldsymbol{x}_*) \leq 0$ **then**
    return $\boldsymbol{x}_*$
  $\mathcal{A} \leftarrow \pi_\theta^\alpha(\mathcal{N}, I, K, \boldsymbol{x}_*)$
  if ANALYZE$(\mathcal{N}, I, K, \mathcal{A})$ = Verified **then**
    return Verified
  $(I_1, I_2) \leftarrow \pi_\theta^I(\mathcal{N}, I, K, \boldsymbol{x}_*)$
  $r_1 \leftarrow$ VERIFY$(\mathcal{N}, I_1, K, \pi_\theta)$
  if $r_1 \neq$ Verified **then**
    return r_1
  return VERIFY$(\mathcal{N}, I_2, K, \pi_\theta)$

At a high-level, the VERIFY procedure works as follows: First, we try to find a counterexample to the given robustness property by solving the optimization problem from Equation 2.1 using the well-known *projected gradient descent (PGD)* technique. If $\mathcal{F}(\boldsymbol{x}_*)$ is non-positive, we have found a true counterexample, so the algorithm produces $\boldsymbol{x}_*$ as a witness to the violation of the property. Otherwise, we try to verify the property using abstract interpretation.

As mentioned in Section 2.1, there are many different abstract domains that can be used to verify the property, and the choice of the abstract domain has a huge impact on the success and efficiency of verification. Thus, our approach leverages the domain policy π_θ^α to choose a sensible abstract domain to use when attempting to verify the property. Specifically, the domain policy

π_θ^α takes as input the network $\mathcal{N}$, the robustness specification (I, K), and the solution $\boldsymbol{x}_*$ to the optimization problem and chooses an abstract domain $\mathcal{A}$ that should be used for attempting to prove the property. If the property can be verified using domain $\mathcal{A}$, the algorithm terminates with a proof of robustness.

In cases where the property is neither verified nor refuted in the current iteration, the algorithm makes progress by splitting the input region I into two disjoint partitions I_1, I_2 such that $I = I_1 \cup I_2$. The intuition is that, even if we cannot prove robustness for the whole input region I, we may be able to increase analysis precision by performing a case split. That is, as long as all points in *both* I_1 and I_2 are classified as having label K, this means that all points in I are also assigned label K since we have $I = I_1 \cup I_2$. In cases where the property is false, splitting the input region into two partition can similarly help adversarial counterexample search because gradient-based optimization methods do not always converge to a global optimum.

Based on the above discussion, the key question is how to partition the input region I into two regions I_1, I_2 so that each of I_1, I_2 has a good chance of being verified or falsified. Since this question again does not have an obvious answer, we use our *partition policy* π_θ^I to make this decision. Similar to the domain policy, π_θ^I takes as input the network, the property, and the solution x_* to the optimization problem and "cuts" I into two sub-regions I_1 and I_2 using a hyper-plane. Then, the property is verified if and only if the recursive call to VERIFY is succsessful on both regions.

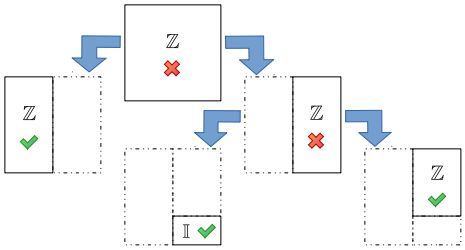


Figure 2.4: The splits chosen for the XOR example.

Example Consider the XOR network from Figure 2.2 and the robustness property $([0.3, 0.7]^2, 1)$. That is, for all inputs $\boldsymbol{x}$ with $0.3 \leq x_1, x_2 \leq 0.7$, $\boldsymbol{x}$ should be assigned to class 1 (assume classes are zero-indexed). We now illustrate how Algorithm 2.1 verifies this property using the plain interval and zonotope abstract domains. The process is illustrated in Figure 2.4, which shows the splits made in each iteration as well as the domain used to analyze each region ($\mathbb{Z}$ denotes zonotopes, and $\mathbb{I}$ stands for intervals).

Algorithm 2.1 starts by searching for an adversarial counterexample, but fails to find one since the property actually holds. Now, suppose that our domain policy π_θ^α chooses zonotopes to try to verify the property. Since the property cannot be verified using zonotopes, the call to Analyze will fail. Thus, we now consult the partition policy π_θ^I to split this region into two pieces $I_1 = [0.3, 0.5] \times [0.3, 0.7]$ and $I_2 = [0.5, 0.7] \times [0.3, 0.7]$.

Next, we recursively invoke Algorithm 2.1 on both sub-regions I_1 and I_2. Again, there is no counterexample for either region, so we use the domain policy to choose an abstract domain for each of I_1 and I_2. Suppose that π_θ^α yields the zonotope domain for both I_1 and I_2. Using this domain, we can

verify robustness in I_1 but not in I_2. Thus, for the second sub-problem, we again consult π_θ^I to obtain two sub-regions $I_{2,1} = [0.5, 0.7] \times [0.3, 0.42]$ and $I_{2,2} = [0.5, 0.7] \times [0.42, 0.7]$ and determine using π_θ^α that $I_{2,1}, I_{2,2}$ should be analyzed using intervals and zonotopes respectively. Since robustness can be verified using these domains, the algorithm successfully terminates. Notice that the three verified subregions cover the entire initial region.

2.3 Learning a Verification Policy

As described in Section 2.2, our decision procedure for checking robustness uses a verification policy $\pi_\theta = (\pi_\theta^\alpha, \pi_\theta^I)$ to choose a suitable abstract domain and an input partitioning strategy. In this section, we discuss our policy representation and how to learn values of θ that lead to good performance.

2.3.1 Policy Representation

In this work, we implement verification policies π_θ^α and π_θ^I using a function of the following shape:

$$\varphi(\theta\, \rho(\mathcal{N}, I, K, \boldsymbol{x}_*)) \tag{2.3}$$

where ρ is a *featurization function* that extracts a feature vector from the input, φ is a *selection* function that converts a real-valued vector to a suitable output (i.e., an abstract domain for π_θ^α and the two subregions for π_θ^I), and θ corresponds to a parameter matrix that is automatically learned from a representative set of training data. We discuss our featurization and selection

functions in this sub-section and explain how to learn parameters θ in the next sub-section.

Featurization As standard in machine learning, we need to convert the input $\iota = (\mathcal{N}, I, K, \boldsymbol{x}_*)$ to a feature vector. Our choice of features is influenced by our insights about the verification problem, and we deliberately use a small number of features for two reasons: First, a large number of dimensions can lead to overfitting and poor generalization (which is especially an issue when training data is fairly small). Second, a high-dimensional feature vector leads to a more difficult learning problem, and contemporary Bayesian optimization engines only scale to a few tens of dimensions.

Concretely, our featurization function considers several kinds of information, including: (a) the behavior of the network near $\boldsymbol{x}_*$, (b) where $\boldsymbol{x}_*$ falls in the input space, and (c) the size of the input space. Intuitively, we expect that (a) is useful because as $\boldsymbol{x}_*$ comes closer to violating the specification, we should need a more precise abstraction, while (b) and (c) inform how we should split the input region during refinement. Since the precision of the analysis is correlated with how the split is performed, we found the same featurization function to work well for both policies π^{α}_{θ} and π^{I}_{θ}. In Section 2.5, we discuss the exact features used in our implementation.

Selection function Recall that the purpose of the selection function φ is to convert $\theta\rho(\iota)$ to a "strategy", which is an abstract domain for π^{α} and a hyper-

plane for π^I. Since the strategies for these two functions are quite different, we use two different selection functions, denoted $\varphi_\alpha, \varphi_I$ for the domain and partition policies respectively.

The selection function φ_α is quite simple and maps $\theta\rho(\iota)$ to a tuple (d, k) where d denotes the base abstract domain (either intervals $\mathbb{I}$ or zonotopes $\mathbb{Z}$ in our implementation) and k denotes the number of disjuncts. Thus, $(\mathbb{Z}, 2)$ denotes the powerset of zonotopes abstract domain, where the maximum number of disjuncts is restricted to 2, and $(\mathbb{I}, 1)$ corresponds to the standard interval domain.

In the case of the partition policy π^I, the selection function φ_I is also a tuple (d, c) where d is the dimension along which we split the input region and c is the point at which to split. In other words, if $\varphi_I(\theta\rho(\iota)) = (d, c)$, this means that we split the input region I using the hyperplane $x_d = c$. Our selection function φ_I does not consider arbitrary hyperplanes of the form $c_1x_1 + \ldots c_nx_n = c$ because splitting the input region along an arbitrary hyperplane may result in sub-regions that are not expressible in the chosen abstract domain. In particular, this is true for both the interval and zonotope domains used in our implementation.

2.3.2 Learning using Bayesian Optimization

As made evident by Equation 2.3, the parameter matrix θ has a huge impact on the choices made by our verification algorithm. However, manually coming up with these parameters is very difficult because (a) the search space

is huge, (b) neural networks lack an easily-interpretable structure, and (c) the right choice of coefficients depends on both the property, the network, and the underlying abstract interpretation engine. In this work, we take a data-driven approach to solve this problem and use *Bayesian optimization* to learn a parameter matrix θ that leads to optimal performance by the verifier on a set of training problems.

Background on Bayesian optimization Given a function $F : \mathbb{R}^n \to \mathbb{R}$, the goal of Bayesian optimization is to find a vector $\boldsymbol{x}^* \in \mathbb{R}^n$ that maximizes F. Importantly, Bayesian optimization does not assume that F is differentiable; also, in practice, it can achieve reasonable performance without having to evaluate F very many times. In our setting, the function F represents the performance of a verification policy. This function is not necessarily differentiable in the parameters of the verification policy, as a small perturbation to the policy parameters can lead to the choice of a different domain. Also, evaluating the function requires an expensive round of abstract interpretation. For these reasons, Bayesian optimization is a good fit to our learning problem.

At a high level, Bayesian optimization repeatedly samples inputs until a time limit is reached and returns the best input found so far. However, rather than sampling inputs at random, the key part of Bayesian optimization is to predict what input is useful to sample next. Towards this goal, the algorithm uses (1) a *surrogate model* $\mathcal{M}$ that expresses our current belief about F, and (b) an *acquisition function* $\mathcal{A}$ that employs $\mathcal{M}$ to decide the most promising

input to sample in the next iteration. The surrogate model $\mathcal{M}$ is initialized to capture prior beliefs about F and is updated based on observations on the sampled points. The acquisition function $\mathcal{A}$ is chosen to trade off exploration and exploitation where "exploration" involves sampling points with high uncertainty, and "exploitation" involves sampling points where $\mathcal{M}$ predicts a high value of F. Given model $\mathcal{M}$ and function $\mathcal{A}$, Bayesian optimization samples the most promising input $\vec{x}$ according to $\mathcal{A}$, evaluates F at $\boldsymbol{x}$, and updates the statistical model $\mathcal{M}$ based on the observation $F(\boldsymbol{x})$. This process is repeated until a time limit is reached, and the best input sampled so far is returned as the optimum. We refer the reader to [72] for a more detailed overview of Bayesian optimization.

Using Bayesian optimization In order to apply Bayesian optimization to our setting, we first need to define what function we want to optimize. Intuitively, our objective function should estimate the quality of the analysis results based on decisions made by verification policy π_θ. Towards this goal, we fix a set S of representative training problems that can be used to estimate the quality of π_θ. Then, given a parameters matrix θ, our objective function F calculates a score based on (a) how many benchmarks in S can be successfully solved within a given time limit, and (b) how long it takes to solve the benchmarks in S. More specifically, our objective function F is parameterized by a time limit $t \in \mathbb{R}$ and penalty $p \in \mathbb{R}$ and calculates the score for a matrix

θ as follows:

$$F(\theta) = -\sum_{s \in S} \text{cost}_\theta(s)$$

where:

$$\text{cost}_\theta(s) = \begin{cases} \text{Time}(\text{Verify}_\theta(s)) & \text{if } s \text{ solved within } t \\ p \cdot t & \text{otherwise} \end{cases}$$

Intuitively, p controls how much we want to penalize failed verification attempts — i.e., the higher the value of p, the more biased the learning algorithm is towards more precise (but potentially slow) strategies. On the other hand, small values of p bias learning towards strategies that yield fast results on the solved benchmarks, even if some of the benchmarks cannot be solved within the given time limit. [3]

In order to apply Bayesian optimization to our problem, we also need to choose a suitable acquisition function and surrogate mode. Following standard practice, we adopt a *Gaussian process* [81] as our surrogate model and use *expected improvement* [17] for the acquisition function.

2.4 Termination and Delta Completeness

In this section, we discuss some theoretical properties of our verification algorithm, including soundness, termination, and completeness. To start with, it is easy to see that Algorithm 2.1 is sound, as it only returns "Verified" once it establishes that *every* point in the input space is classified as K. This is the case because every time we split the input region I into two sub-regions

[3] In our implementation, we choose $p = 2$, $t = 700s$.

I_1, I_2, we ensure that $I = I_1 \cup I_2$, and the underlying abstract interpreter is assumed to be sound. However, it is less clear whether Algorithm 2.1 always terminates or whether it has any completeness guarantees.

Our first observation is that the VERIFY procedure, *exactly* as presented in Algorithm 2.1, does not have termination guarantees under realistic assumptions about the optimization procedure used for finding adversarial counterexamples. Specifically, if the procedure MINIMIZE invoked in Algorithm 2.1 returned a *global* minimum, then we could indeed guarantee termination. [4] However, since gradient-based optimization procedures do not have this property, Algorithm 2.1 may not be able to find a true adversarial counterexample even as we make the input region infinitesimally small. Fortunately, we can guarantee termination and a form of completeness (known as δ-completeness) by making a very small change to Algorithm 2.1.

To guarantee termination, we will make the following slight change to Algorithm 2.1: Rather than checking $\mathcal{F}(\boldsymbol{x}_*) \leq 0$ (for $\mathcal{F}$ as defined in Equation 2.2) we will instead check:

$$\mathcal{F}(\boldsymbol{x}_*) \leq \delta \tag{2.4}$$

for some chosen $\delta > 0$. While this modification can cause our verification algorithm to produce false positives under certain pathological conditions, the analysis can be made as precise as necessary by picking a value of δ that

[4] However, if we make this assumption, the optimization procedure itself would be a sound and complete decision procedure for verifying robustness!

is arbitrarily close to 0. Furthermore, under this change, we can now prove termination under some mild and realistic assumptions. In order to formally state these assumptions, we first introduce the following notion of the *diameter* of a region:

Definition 2.1. For any set $X \subseteq \mathbb{R}^n$, its *diameter* $D(X)$ is defined as

$$D(X) = \sup\{\|\boldsymbol{x}_1 - \boldsymbol{x}_2\|_2 \mid \boldsymbol{x}_1, \boldsymbol{x}_2 \in X\}$$

if this value exists. Otherwise the set is said to have infinite diameter.

We now use this notion of diameter to state two key assumptions that are needed to prove termination:

Assumption 2.1. There exists some $\lambda \in (0, 1)$ such that for any network $\mathcal{N}$, input region I, and point $\boldsymbol{x}_* \in I$, if $\pi^I(N, I, \boldsymbol{x}_*) = (I_1, I_2)$, then $D(I_1) < \lambda D(I)$ and $D(I_2) < \lambda D(I)$.

Intuitively, this assumption states that the two resulting subregions after splitting are smaller than the original region by some factor λ. It is easy to enforce this condition on any partition policy by choosing a hyper-plane $x_d = c$ where c is not at the boundary of the input region.

Our second assumption concerns the abstract domain:

Assumption 2.2. Let $\mathcal{N}^\#$ be the abstract transformer representing a network $\mathcal{N}$. For a given input region I, we assume there exists some $K_\mathcal{N} \in \mathbb{R}$ such that $D(\gamma(\mathcal{N}^\#(\alpha(I)))) < K_\mathcal{N} D(I)$.

This assumption asserts that the Lipschitz continuity of the network extends to its abstract behavior. Note that this assumption holds in several numerical domains including intervals, zonotopes, and powersets thereof.

Theorem 2.1. *Consider the variant of Algorithm 2.1 where the predicate is replaced with Eq. 2.4 as described above. Then, if the input region has finite diameter, the verification algorithm always terminates under Assumptions 2.1 and 2.2.*

Proof. To improve readibility, we define $F(I_k) = \gamma(\mathcal{N}^{\#}(\alpha(I_k)))$.

By Assumption 2.1 there exists some $\lambda \in \mathbb{R}$ with $0 < \lambda < 1$ such that for any input region I', splitting I' with Refine yields regions I'_1 and I'_2 with $D(I'_1) < \lambda D(I')$ and $D(I'_2) < \lambda D(I')$. Because there is one split for each node in the recursion tree, at a recursion depth of k, the region I_k under consideration has diameter $D(I_k) < \lambda^k D(I)$. By Assumption 2.2, there exists some $K_{\mathcal{N}}$ such that $D(F(I_k)) < K_{\mathcal{N}} D(I_k)$. Notice that when

$$k > \log_\lambda \left(\frac{\delta}{2K_{\mathcal{N}} D(I)} \right)$$

we must have $D(F(I_k)) < \delta/2$.

We will now show that when k satisfies the preceding condition, Algorithm 2.1 must terminate without recurring. In this case, suppose $\boldsymbol{x}_*$ is the point returned by the call to Minimize and the algorithm does not terminate. Then $\mathcal{F}(\boldsymbol{x}_*) > \delta$ and in particular, $(\mathcal{N}(\boldsymbol{x}_*))_K - (\mathcal{N}(\boldsymbol{x}_*))_i > \delta$. Since Analyze is sound, we must have $\mathcal{N}(\boldsymbol{x}_*) \in F(I_k)$. Then since $D(F(I_k)) < \delta/2$, we must

have that for any point $\boldsymbol{y}' \in F(I_k)$, $\|\mathcal{N}(\boldsymbol{x}_*) - \boldsymbol{y}'\|_2 < \delta/2$. In particular, for all i, $|(\mathcal{N}(\boldsymbol{x}_*))_i - \boldsymbol{y}'_i| < \delta/2$, so $\boldsymbol{y}'_i > (\mathcal{N}(\boldsymbol{x}_*))_i - \delta/2$ and $\boldsymbol{y}'_i < (\mathcal{N}(\boldsymbol{x}_*))_i + \delta/2$. Then, for all i,

$$\begin{aligned}
\boldsymbol{y}_K - \boldsymbol{y}_i &> ((\mathcal{N}(x_*))_K - \delta/2) - ((\mathcal{N}(x_*))_i + \delta/2) \\
&= ((\mathcal{N}(x_*))_K - (\mathcal{N}(x_*))_i) - \delta \\
&> 0
\end{aligned}$$

Thus, for each point $\boldsymbol{y}' \in F(I_k)$, we have $\boldsymbol{y}'_K > \boldsymbol{y}'_i$. Since $\boldsymbol{y}'$ ranges over the *overapproximated* output produced by the abstract interpreter, this exactly satisfies the condition which ANALYZE is checking, so ANALYZE must return Verified. Therefore, the maximum recursion depth of Algorithm 2.1 is bounded, so it must terminate. □

In addition to termination, our small modification to Algorithm 2.1 also ensures a property called δ-completeness [32]. In the context of satisfiability over real numbers, δ-completeness means that, when the algorithm returns a satisfying assignment σ, the formula is either indeed satisfiable or a δ-perturbation on its numeric terms would make it satisfiable. To adapt this notion of δ-completeness to our context, we introduce the folowing concept δ-counterexamples:

Definition 2.2. For a given network $\mathcal{N}$, input region I, target class K, and $\delta > 0$, a δ-counterexample is a point $x \in I$ such that for some j with $1 \leq j \leq m$ and $j \neq K$, $(\mathcal{N}(x))_K - (\mathcal{N}(x))_j \leq \delta$.

Intuitively, a δ-counterexample is a point in the input space for which the output almost violates the given specification. We can view δ as a parameter which controls how close to violating the specification a point must be to be considered "almost" a counterexample.

Theorem 2.2. *Consider the variant of Algorithm 2.1 where the predicate at replaced with Equation 2.4. Then, the verification algorithm is δ-complete — i.e., if the property is not verified, it returns a δ-counterexample.*

Proof. First note that by Theorem 2.1, Algorithm 2.1 must terminate. Therefore the algorithm must return some value, and we can prove this theorem by analyzing the possible return values. We only care about the case where the algorithm does not return "Verified" so we can ignore the case where the property is verified by abstract interpretation. The return in the first if statement is only reached after checking that $\boldsymbol{x}_*$ is a δ-counterexample, so clearly if that return statement is used then the algorithm returns a δ-counterexample. This leaves the two recursive calls. We suppose by induction that the recursive calls are δ-complete. Then if r_1 is not Verified, it must be a δ-counterexample. Thus the return statement which returns r_1 also returns a δ-counterexample. Similarly, if r_2 is not Verified, then it is a δ-counterexample, so the last return statement returns a δ-counterexample. □

2.5 Implementation

We have implemented the ideas proposed in this paper in a tool called CHARON, written in C++. Internally, CHARON uses the ELINA abstract interpretation library [1] to implement the ANALYZE procedure from Algorithm 2.1, and it uses the BayesOpt library [69] to perform Bayesian optimization.

Parallelization Our proposed verification algorithm is easily parallelizable, as different calls to the abstract interpreter can be run on different threads. Our implementation takes advantage of this observation and utilizes as many threads as the host machine can provide by running different calls to ELINA in parallel.

Training We trained our verification policy on 12 different robustness properties of a neural network used in the ACAS Xu collision avoidance system [50]. However, since even verifying even a single benchmark can take a very long time, our implementation uses two tactics to reduce training time. First, we parallelize the training phase of the algorithm using the MPI framework [29] and solve each benchmark at the same time. Second, we set a time limit of 700 seconds (per-process cputime) per benchmark. Contrary to what we may expect from machine learning systems, a small set of benchmarks is sufficient to learn a good strategy for our setting. We conjecture that this is because the relatively small number of features allowed by Bayesian optimization helps to

regularize the learned policy.

Featurization Recall that our verification policy uses a *featurization function* to convert its input to a feature vector. As mentioned in Section 2.3, this featurization function should select a compact set of features so that our training is efficient and avoids overfitting our policy to the training set. These features should also capture revalant information about the network and the property so that our learned policy can generalize across networks. With this in mind, we used the following features in our implementation:

- the distance between the center of the input region I and the solution $\boldsymbol{x}_*$ to the optimization problem,
- the value of the objective function $\mathcal{F}$ (Equation 2.2) at $\boldsymbol{x}_*$,
- the magnitude of the gradient of the network at $\boldsymbol{x}_*$, and
- average length of the input space along each dimension.

Selection Recall from Section 2.3 that our verification policy π uses two different selection functions φ^{α} and φ^{I} for choosing an abstract domain and splitting plane respectively.

The selection function φ^{I} takes a vector of three inputs. The first two are real-valued numbers that decide which dimension to split on. Rather than considering all possible dimensions, our implementation chooses between two

dimensions to make training more manageable. The first one is the longest dimension (i.e., input dimension with the largest length), and the second one is the dimension that has the largest *influence* [100] on $(\mathcal{N}(x))_K$. The last input to the selection function is the offset at which to split the region. This value is clipped to $[0, 1]$ and then interpreted as a ratio of the distance from the center of the input region I to the solution $\boldsymbol{x}_*$ of Equation 2.1. For example, if the value is 0, the region will be bisected, and if the value is 1, then the splitting plane will intersect $\boldsymbol{x}_*$. Finally, if the splitting plane is at the boundary of I, it is offset slightly so that the strategy satisfies Assumption 2.1.

The selection function φ^α for choosing an abstract domain takes a vector of two inputs. The first controls the base abstract domain (intervals or zonotopes) and the second controls the number of disjuncts to use. In both cases, the output is extracted by first clipping the input to a fixed range and then discretizing the resulting value.

2.6 Evaluation

To evaluate the ideas proposed in this paper, we conduct an experimental evaluation that is designed to answer the following three research questions:

(RQ1) How does Charon compare against state-of-the-art tools for proving neural network robustness?

(RQ2) How does counterexample search impact the performance of Charon?

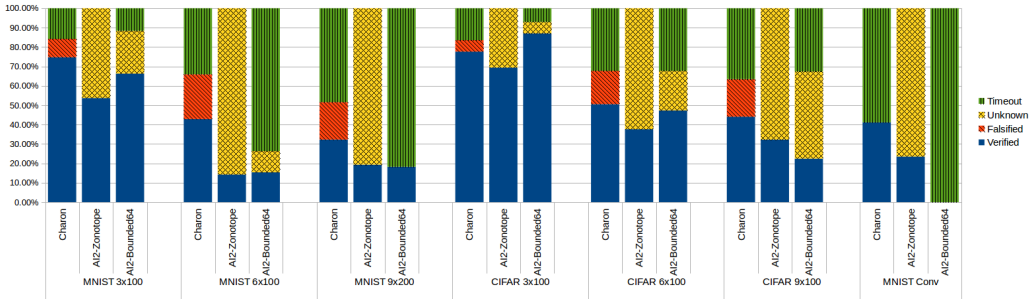


Figure 2.5: Summary of results for AI^2 and CHARON.

(RQ3) What is the impact of learning a verification policy on the performance of CHARON?

Benchmarks To answer these research questions, we collected a benchmark suite of 602 verification problems across 7 deep neural networks, including one convolutional network and several fully connected networks. The fully connected networks have sizes 3×100, 6×100, 9×100, and 9×200, where $N\times M$ means there are N fully connected layers and each interior layer has size M. The convolutional network has a LeNet architecture [59] consisting of two convolutional layers, followed by a max pooling layer, two more convolutional layers, another max pooling layer, and finally three fully connected layers. All of these networks were trained on the MNIST [59] and CIFAR [54] datasets.

2.6.1 Comparison with AI^2 (RQ1)

For each network, we attempt to verify around 100 robustness properties. Following prior work [35], the evaluated robustness properties are so-called *brightening attacks* [77]. For an input point $\boldsymbol{x}$ and a threshold τ, a

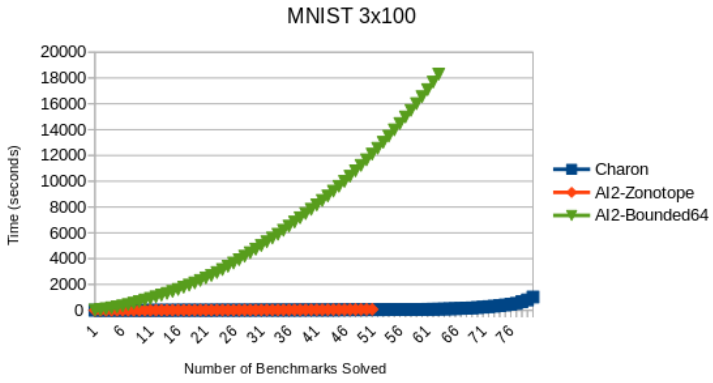


Figure 2.6: Comparison on a 3×100 MNIST network.

brightening attack consists of the input region

$$I = \{\boldsymbol{x}' \in \mathbb{R}^n \mid \forall i.(\boldsymbol{x}_i \geq \tau \wedge \boldsymbol{x}_i \leq \boldsymbol{x}'_i \leq 1) \vee \boldsymbol{x}'_i = \boldsymbol{x}_i\}\,.$$

That is, for each pixel in the input image, if the value of that pixel is greater than τ, then the corresponding pixel in the perturbed image may be anywhere between the initial value and one, and all other pixels remain unchanged.

Setup All experiments described in this section were performed on the Google Compute Engine (GCE) [2] using an 8 vcpu instance with 10.5 GB of memory. All time measurements report the total CPU time (rather than wall clock time) in order to avoid biasing the results because of CHARON's parallel nature. For the purposes of this experiment, we set a time limit of 1000 seconds per benchmark.

In this section we compare CHARON with AI^2 [5], a state-of-the-art tool

[5]Because we did not have access to the original AI^2, we reimplemented it. However, to

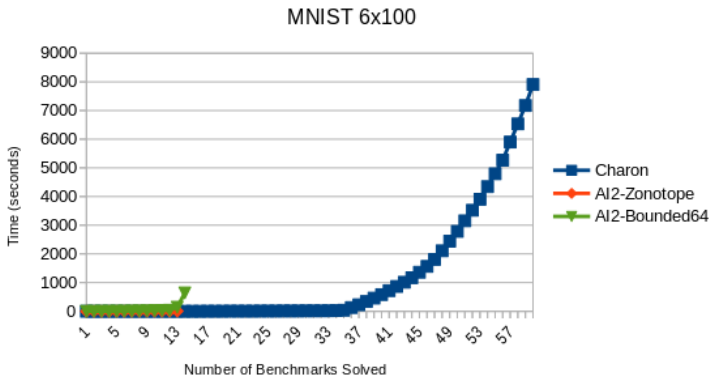


Figure 2.7: Comparison on a 6×100 MNIST network.

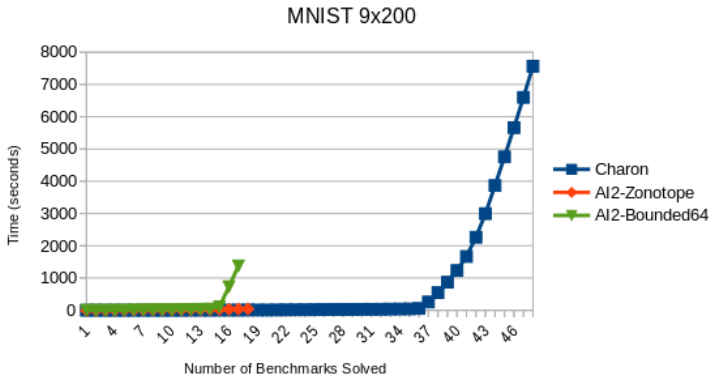


Figure 2.8: Comparison on a 9×200 MNIST network.

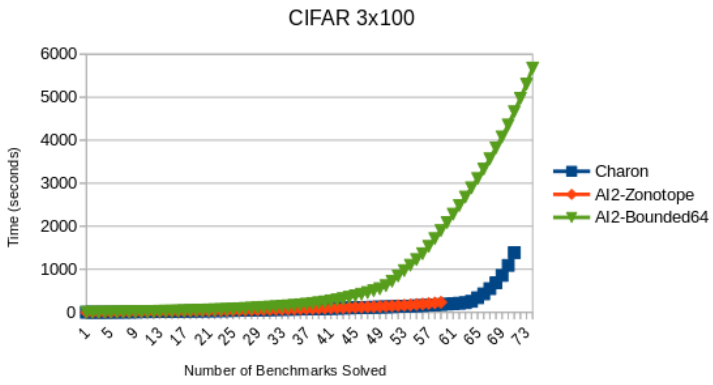


Figure 2.9: Comparison on a 3×100 CIFAR network.

for verifying network robustness [35]. As discussed in Section 2.1, AI^2 is incomplete and requires the user to specify which abstract domain to use. Following their evaluation strategy from the IEEE S&P paper [35], we instantiate AI^2 with two different domains, namely zonotopes and bounded powersets of zonotopes of size 64. We refer to these two variants as AI^2-Zonotope and AI^2-Bounded64.

The results of this comparison are summarized in Figure 2.5. This graph shows the percentage of benchmarks each tool was able to verify or falsify, as well as the percentage of benchmarks where the tool timed out and the percentage where the tool was unable to conclude either true or false. Note that, because CHARON is δ-complete, there are no "unknown" results for it, and because AI^2 cannot find counterexamples, AI^2 has no "falsified" results.

allow for a fair comparison, we use the same underlying abstract interpretation library, and we implement the transformers exactly as described in [35].

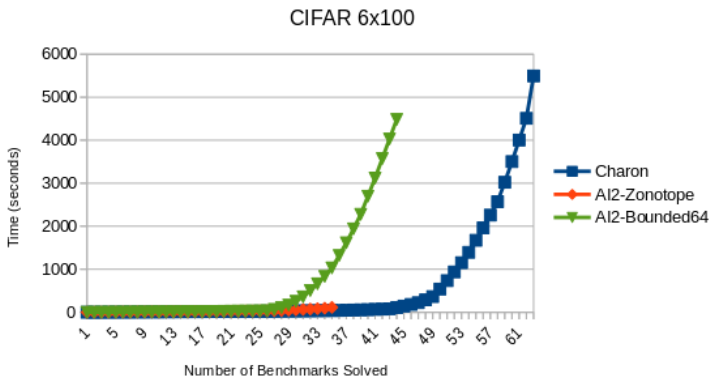


Figure 2.10: Comparison on a 6×100 CIFAR network.

The details for each network are shown in Figures 2.6 - 2.12. Each chart shows the cumulative time taken on the y-axis and the number of benchmarks solved on the x-axis (so lower is better). The results for each tool include only those benchmarks that the tool could solve correctly within the time limit of 1000 seconds. Thus, a line extending further to the right indicates that the tool could solve more benchmarks. Since AI^2-Bounded64 times out on every benchmark for the convolutional network, it does not appear in Figure 2.12.

The key take-away lesson from this experiment is that CHARON is able to both solve more benchmarks compared to AI^2-Bounded64 on most networks, and it is able to solve them much faster. In particular, CHARON solves 59.7% (resp. 84.7%) more benchmarks compared to AI^2-Bounded64 (resp. AI^2-Zonotope). Furthermore, among the benchmarks that can be solved by both tools, CHARON is 6.15× (resp. 1.12×) faster compared to AI^2-Bounded64 (resp. AI^2-Zonotope). Thus, we believe these results demonstrate the advan-

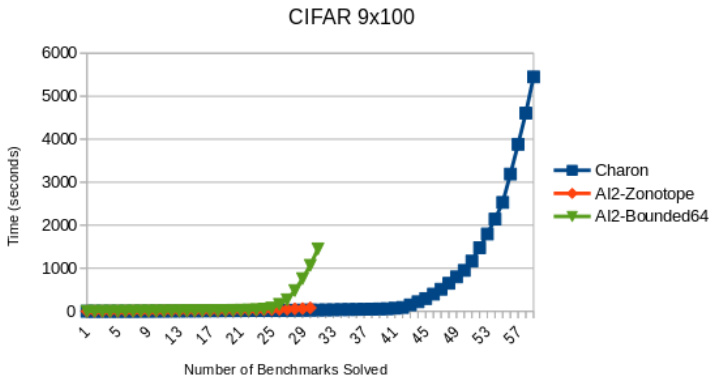


Figure 2.11: Comparison on a 9×100 CIFAR network.

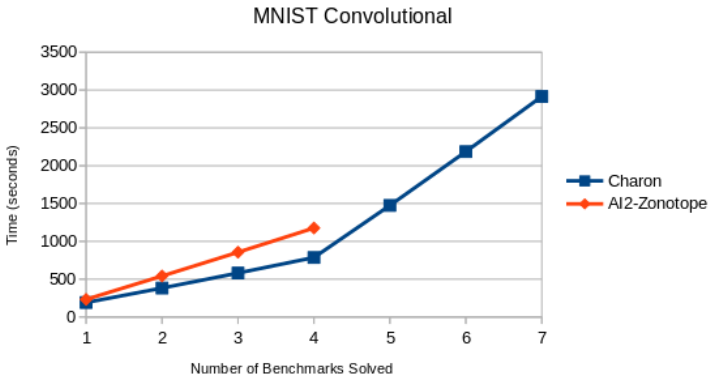


Figure 2.12: Comparison on a convolutional network.

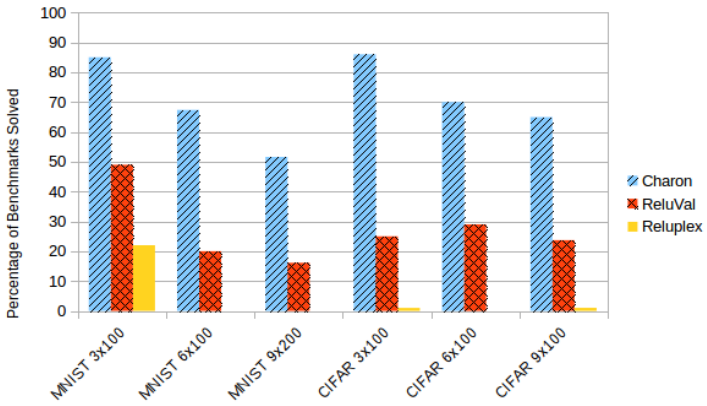


Figure 2.13: Comparison with ReluVal.

tages of our approach compared to AI^2.

2.6.2 Comparison with Complete Tools (RQ1)

In this section we compare Charon with other complete tools for robustness analysis, namely ReluVal [100] and Reluplex [51]. Among these tools, Reluplex implements a variant of Simplex with built-in support for the ReLU activation function [51], and ReluVal is an abstraction refinement approach without learning or counterexample search.

To perform this experiment, we evaluate all three tools on the same benchmarks from Section 2.6.1. However, since ReluVal and Reluplex do not support convolutional layers, we exclude the convolutional network from this evaluation.

The results of this comparison are summarized in Figure 2.13. Across all benchmarks, Charon is able to solve 2.6× (resp. 16.6×) more problems

compared to ReluVal (resp. Reluplex). Furthermore, it is worth noting that the set of benchmarks that can be solved by Charon is a strict superset of the benchmarks solved by ReluVal.

2.6.3 Impact of Counterexample Search (RQ2)

To understand the benefit of using optimization to search for counterexamples, we now compare the number of properties that can be *falsified* using Charon vs. Reluplex and ReluVal. (Recall that AI^2 is incomplete and cannot be used for falsification.) Among the 585 benchmarks used in the evaluation from Section 2.6.2, Charon can falsify robustness of 123 benchmarks. In contrast, Reluplex can only falsify robustness of one benchmark, and ReluVal cannot falsify any of them. Thus, we believe these results demonstrate the usefulness of incorporating optimization-based counterexample search into the decision procedure.

2.6.4 Impact of Learning a Verification Policy (RQ3)

Recall that a key feature of our algorithm is the use of a machine-learnt verification policy π to choose a refinement strategy. To explore the impact of this design choice, we compare our technique against ReluVal on the subset of the 585 benchmarks for which the robustness property holds. In particular, as mentioned earlier, ReluVal is also based on a form of abstraction refinement but uses a static, hand-crafted strategy rather than one that is learned from data. Thus, comparing against ReluVal on the verifiably-robust bench-

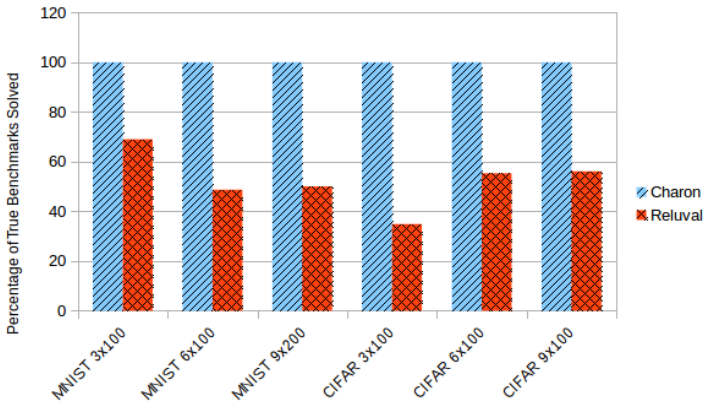


Figure 2.14: Comparison with ReluVal on verified benchmarks.

marks allows us to evaluate the benefits of learning a verification policy from data. [6]

The results of this comparison are shown in Figure 2.14. As we can see from this figure, ReluVal is still only able to solve between 35% and 70% of the benchmarks that can be successfully solved by Charon. Thus, these results demonstrate that our data-driven approach to learning verification policies is useful for verifying network robustness.

[6]We compare with ReluVal directly rather than reimplementing the ReluVal strategy inside Charon because our abstract interpretation engine does not support the domain used by ReluVal. Given this, we believe the comparison to ReluVal is the most fair available option.

Chapter 3

Safe RL Background and Notation

In this chapter, we will present some background material which is common to the techniques discussed in the rest of the dissertation.

3.1 Reinforcement Learning

Throughout this dissertation, we will be developing techniques to control systems which have both safety constraints and performance objectives, formalized as constrained Markov decision processes (CMDP's). A CMDP is a tuple $\mathcal{M} = (\mathcal{S}, \mathcal{A}, P, p_0, r, c)$. Here $\mathcal{S}$ is a set of states, $\mathcal{A}$ is a set of actions, $P(\boldsymbol{s}' \mid \boldsymbol{s}, \boldsymbol{a})$ is a transition distribution, p_0 is a distribution of initial states, $r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is an immediate reward signal, and $c : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is an immediate cost function. We will often refer to the CMDP $\mathcal{M}$ as the "environment" in which an agent is operating. For the purposes of this dissertation, we will assume c is a boolean indicator depending only on states. That is, there exists a distinguished set $\mathcal{S}_U \subset \mathcal{S}$ such that $c(\boldsymbol{s}, \boldsymbol{a})$ is 1 if taking action $\boldsymbol{a}$ from $\boldsymbol{s}$ leads to a state $\boldsymbol{s}' \in \mathcal{S}_U$ and 0 otherwise. In general, we assume that $\mathcal{S}$, $\mathcal{A}$, and $\mathcal{S}_U$ are known, where P, p_0, and r are not. However in subsequent chapters we will look at settings where we have different amounts of prior knowledge

about P and p_0.

We will also need a number of other definitions to state the safe learning and safe exploration problems:

- A *policy*, $\pi(\boldsymbol{a} \mid \boldsymbol{s})$ for $\boldsymbol{a} \in \mathcal{A}$ and $\boldsymbol{s} \in \mathcal{S}$ is a function which takes a state and yields a distribution over actions.
- A policy π can be used to generate *trajectories*, which are sequences of states and actions $\boldsymbol{s}_0, \boldsymbol{a}_0, \ldots, \boldsymbol{s}_n, \boldsymbol{a}_n$ such that $\boldsymbol{s}_0 \sim p_0$, $\boldsymbol{a}_i \sim \pi(\cdot \mid \boldsymbol{s}_i)$, and $\boldsymbol{s}_{i+1} \sim P(\cdot \mid \boldsymbol{s}_i, \boldsymbol{a}_i)$.
- A policy π induces a distribution P_π over trajectories.
- We will choose a *discount factor* $\gamma < 1$.
- The *return* of a trajectory $\tau = \boldsymbol{s}_0, \boldsymbol{a}_0, \ldots, \boldsymbol{s}_n, \boldsymbol{a}_n$ is
$$R(\tau) = \sum_{i=0}^{n} \gamma^i r(\boldsymbol{s}_i, \boldsymbol{a}_i).$$
- The (undiscounted) *cost* of a trajectory $\tau = \boldsymbol{s}_0, \boldsymbol{a}_0, \ldots, \boldsymbol{s}_n, \boldsymbol{a}_n$ is
$$C(\tau) = \sum_{i=0}^{n} c(\boldsymbol{s}_i, \boldsymbol{a}_i).$$
- The *expected return* of a policy π is $R(\pi) = \mathbb{E}_{\tau \sim P_\pi}[R(\tau)]$.
- The *expected cost* of a policy π is $C(\pi) = \mathbb{E}_{\tau \sim P_\pi}[C(\tau)]$.

For the purposes of this work, we will assume the cost signal indicates *catastrophic* failure, meaning that a trajectory ends as soon as an unsafe state is

encountered. This implies that $C(\tau)$ is itself a boolean indication of safety over entire trajectories. As a result, $C(\pi)$ gives the probability that a trajectory samplied according to π will result in unsafe behavior.

With all this in place, we can define the goal of reinforcement learning: find a policy π^* which maximizes the expected return. That is, solve

$$\pi^* = \arg\max_{\pi} R(\pi). \tag{3.1}$$

However, this definition does not consider safety at all. The goal of *safe* reinforcement learning is to find

$$\begin{aligned} \pi^* &= \arg\max_{\pi} R(\pi) \\ \text{s.t. } & C(\pi) \leq \delta \end{aligned} \tag{3.2}$$

for some predefined safety level δ. Intuitively, the cost constraint in Equation 3.2 requires that the probability of safety violations under the policy π is below some chosen threshold. If $\delta = 0$, so that no possibility of safety violation is allowed at all, then we call this *verified* RL.

All of these problems so far have only dealt with the safety of the final policy. In practice, however, reinforcement learning algorithms work by executing a sequence of policies in the environment in order to collect reward data and generate future policies. In many cases, we need to consider the safety of not only the final policy, but also *every* policy which is executed in the environment during training. This gives rise to the problem of *safe exploration*. We formalize this problem in terms of a *learning process* which is

a sequence of policies $\mathcal{L} = \pi_0, \ldots, \pi_n$ which holds all of the policies executed during training by a reinforcement learning algorithm. The safe exploration problem is to solve Equation 3.2 while additionally ensuring that each policy in $\mathcal{L}$ is safe, i.e., $C(\pi_i) \leq \delta$ for all i.

3.2 Model-Based RL

At a high-level, all of the techniques presented in this dissertation are built on top of *model-based RL* [21, 49, 92, 93]. Model-based RL is a class of approaches to the reinforcement learning problem in which we explicit develop (or are given) a *model* of the environment. Recall from Section 3.1 that we usually assume the transition distribution P, the initial distribution p_0, and the reward function r are unknown. In model-based RL, the high-level technique is to collect data from the environment and use that data to learn approximations $\hat{P}$, $\hat{p_0}$, and $\hat{r}$. These approximations are then used to support policy optimization.

3.3 Shielding and Neurosymbolic Learning

All of the techniques discussed in this dissertation are based on *shielding*. Shielding is an approach to ensuring the safety of a controlled dynamical system via the use of a *shield*, which can be thought of as a fallback controller. The basic idea is to develop two separate controllers. One is known to be safe, but may not achieve particularly high performance, while the other is performant but may not always be safe. In addition to these two controllers, we also

develop a *monitor* which can analyze the action proposed by the performance controller and determine whether that action is safe. Then, every time we need to choose an action, we first ask the performant controller to generate one. We then use the monitor to check whether the proposed action is safe. If it is, we can use that action in order to get the highest possible performance. Otherwise, we get an action from the shield which will guarantee safety.

Formally, let π_S be a safe (resp. verified) policy and let π_N but a performant, but potentially unsafe, policy. (We use the subscript N because in this dissertation the performant policies are neural networks.) Now we also need a monitor. The monitor is a boolean function $M : \mathcal{S} \times \mathcal{A} \to \{0, 1\}$. The key property of monitors is $M(\boldsymbol{s}, \boldsymbol{a})$ is true if, when we take action $\boldsymbol{a}$ in state $\boldsymbol{s}$ and then *always use π_S in future time steps* the resulting trajectory will be safe (resp. verified). Intuitively, the monitor determines whether, after taking the proposed action, the safe policy π_S can be used to recover to a safe trajectory. How exactly this is accomplished varies in the different techniques presented in the rest of this dissertation.

In this dissertation, the performant policy will always be a neural network, while the shield policy may be either a neural network or a "symbolic" policy. In this context, the term "symbolic" indicates that the policy has some interpretable internal structure which can be used to support anlysis and verification. Concretely, these symbolic polices take the form of a traditional program with traditional programming constructs (conditionals, loops, etc.). Given a neural policy, a shield, and a monitor, we will often consider them

together as a single *neurosymbolic* policy.

3.4 Mirror Descent

The safe exploration techniques described in this dissertation are all based on a form of *mirror descent* in the policy space, an idea first pioneered in [99]. In our context, mirror descent can be thought of as a way to solve optimization problems when gradients are difficult to compute directly. It relies on two related classes of policies: $\mathcal{G}$ which contains shield policies and $\mathcal{H}$ which consists of neurosymbolic policies. To formalize these classes, let $\mathcal{F}$ be a predefined class of neural policies (i.e., $\mathcal{F}$ consists of the set of all possible neural networks with some fixed architecture). We then assume policies in $\mathcal{H}$ consist of some combination of a shield from $\mathcal{G}$ with a neural network from $\mathcal{F}$.

The goal of mirror descent is to find the optimal *symbolic* policy $\pi_S^* \in \mathcal{G}$. In order to do this, we repeatedly apply three steps:

1. LIFT: Given a symbolic policy π_S generate a neurosymbolic policy π which has the same behavior.

2. UPDATE: Use approximate gradient descent in the neurosymbolic policy space to train π to a higher-performance policy π'.

3. PROJECT: Find a symbolic policy π'_S which is as similar as possible to π' under a given distance metric D.

The exact process for each of these three steps varies between the differnt

techniques outlined in this dissertation. Once we have defined these three procedures, under certain assumptions we can prove that mirror descent will converge to the optimal symbolic policy.

Chapter 4

Safe Exploration in Known Environments[1]

In this chapter, we improve the state of the art in safe exploration through an RL framework, called REVEL[2], that allows learning over continuous state and action spaces, supports (partially) neural policy representations and contemporary policy gradient methods for learning, while also ensuring that every intermediate policy that the learner constructs during exploration is safe on worst-case inputs. Like previous efforts, REVEL uses monitoring and shielding. However, unlike in prior work, the monitor and the shield are updated as learning progresses.

A key feature of our approach is that we repeatedly invoke a formal verifier from within the learning loop. Doing this is challenging because of the high computational cost of verifying neural networks. We overcome this challenge using a neurosymbolic policy representation in which the shield and the monitor are expressed in an easily-verifiable symbolic form, whereas the normal-mode policy is given by a neural network. Overall, this representation

[1]This chapter is based on [8]. The author of this dissertation was responsible for main idea along with the majority of the theoretical analysis and implementation.

[2]REVEL stands for **Re**inforcement learning with **ve**rified exp**l**oration. The current implementation is available at `https://github.com/gavlegoat/safe-learning`.

admits efficient gradient-based learning as well as efficient updates to both the shield and monitor.

To learn such neurosymbolic policies, we build on PROPEL [99], a recent RL framework in which policies are represented in compact symbolic forms (albeit without consideration of safety), and design a learning algorithm that performs a functional mirror descent in the space of neurosymbolic policies. The algorithm views the set of shields as being obtained by imposing a constraint on the general policy space. Starting with a safe but suboptimal shield, it alternates between: (i) safely *lifting* the current shield into the unconstrained policy space by adding a neural component; (ii) safely *updating* this neurosymbolic policy using approximate gradients; and (iii) using a form of imitation learning to *project* the updated policy back into the constrained space of shields. Importantly, none of these steps requires direct verification of neural networks.

Our empirical evaluation, on a suite of continuous control problems, shows that REVEL enforces safe exploration in many scenarios where established RL algorithms (including CPO [3], which is motivated by safe RL) do not, while discovering policies that outperform policies based on static shields. Also, building on results for PROPEL, we develop a theoretical analysis of REVEL.

In summary, this chapter presents the following contributions:

- We introduce the first RL approach to use deep policy representations

and policy gradient methods while guaranteeing formally verified exploration.

- We propose a new solution to this problem which combines ideas from RL and formal methods, and we show that our method has convergence guarantees.

- We present promising experimental results for our method in the continuous control setting.

4.1 Preliminaries

4.1.1 Formally Verified Exploration

Formally, we represent our knowledge of the environment with an approximate transition function $P^{\#} : \mathcal{S} \times \mathcal{A} \to 2^{\mathcal{S}}$ along with a set of initial states $\mathcal{S}_0$. The transition function satisfies $P^{\#}(\boldsymbol{s}, \boldsymbol{a}) \supseteq \text{supp}(P(\cdot \mid \boldsymbol{s}, \boldsymbol{a}))$. Intuively, this means that any possible behavior of the environment is captured by the approximate transition. Additionally, the initial states satisfy $\mathcal{S}_0 \supseteq \text{supp}(p_0)$. That is, the set of initial states is includes every possible initial state that might be chosen for the real environment.

Taken together, the two approximations $P^{\#}$ and $\mathcal{S}_0$ effectively "forget" the probabilistic nature of the environment and convert it to an unspecified nondeterminism. This is appropriate in the context of *verified* exporation because we are concerned with whether or not the system can *ever* behave unsafely regardless of how likely or unlikely that behavior is.

Given these approximate dynamics, we will inductively define finite-horizon reachability for a given policy π and starting set $S \subseteq \mathcal{S}$:

$$\text{reach}_1(\pi, S) = \bigcup_{\boldsymbol{s} \in S, \boldsymbol{a} \in \text{supp}(\pi(\cdot|\boldsymbol{s}))} P^{\#}(\boldsymbol{s}, \boldsymbol{a})$$

$$\text{reach}_{i+1}(\pi, S) = \text{reach}_1(\pi, \text{reach}_i(\pi, S))$$

We can now present an equivalent definition of verified learning which will be easier to work with in this section: a policy π is *verified* if $\mathcal{S}_U \cap \bigcup_i \text{reach}_i(\pi, \mathcal{S}_0) = \varnothing$. We use the notation $\text{Verf}(\pi)$ to indicate that π is verified. Then the verified exploration problem over a learning process $\mathcal{L} = \pi_0, \ldots, \pi_n$ can be restated as:

$$\pi_n = \underset{\pi \text{ s.t. } \text{Verf}(\pi)}{\arg\max} R(\pi) \tag{4.1}$$

$$\text{s.t. } \forall 0 \leq i \leq n : \text{Verf}(\pi_i) \tag{4.2}$$

4.1.2 Inductive Invariants and Abstract Interpretation

The learning algorithm presented in Section 4.2 relies on an oracle for the formal verification of policies. Given a policy π, this oracle attempts to construct a proof of the property $\text{Verf}(\pi)$. This proof takes the form of an *inductive invariant*, defined as a set of states ϕ such that:

1. ϕ includes the initial states, $\mathcal{S}_0 \subseteq \phi$;
2. ϕ is closed under the approximate transition relation, $\text{reach}_1(\pi, \phi) \subseteq \phi$; and
3. ϕ does not include any unsafe states, $\phi \cap \mathcal{S}_U = \varnothing$.

The second condition here is critical: it ensures that once the system is within ϕ, it can never leave under any possible behavior of the environment. Combined with the first condition (which ensures that we start in ϕ) and the third condition (which ensures that no states in ϕ are unsafe), this gives a proof that the system cannot encounter any unsafe states.

There are many ways to construct inductive invariants. In this work, we use *abstract interpretation* [22], which maintains an *abstract* state that approximates the *concrete* states which the system can reach. An abstract state is essentially a set of states which has some kind of structure to make computations over the entire set feasible. For example, an abstract state might consist of hyperinterval in the state space, which defines independent bounds on each state variable. Critically, the abstract state is an *overapproximation*, meaning that every reachable state is represented in the abstract state. However, the abstract state may also include some states which are not reachable in reality because we need to make approximations in order to make the analysis tractable.

In order to apply abstract interpretation to the verified exploration problem, we start with an abstract state representing the initial states $\mathcal{S}_0$. This abstract state is then propagated through the approximated environment transition $P^{\#}$ in order to build up approximations of $\text{reach}_i(\pi, \mathcal{S}_0)$ for increasing i. Then, if the *abstract* state does not include any unsafe states, we can conclude that all possible *concrete* states will be safe as well.

4.2 Learning Algorithm

Algorithm 4.1 Reinforcement Learning with Formally Verified Exploration (REVEL)

Input: Symbolic Policy Class $\mathcal{G}$ & Neural Policy Class $\mathcal{F}$.
Input: Initial $\pi_S^{(0)} \in \mathcal{G}$, with the guarantee $\phi_0 \vdash \text{Verf}(\pi_S^{(0)})$ for some ϕ_0
Define class $\mathcal{H} = \{\pi(s) \equiv \text{if } P^{\#}(s, \pi_N(s)) \subseteq \phi \text{ then } \pi_N(s) \text{ else } \pi_S(s)\}$
for $t = 1, \ldots, T$ **do**
  $\pi^{(t)} \leftarrow \textsc{Lift}(\pi_S^{(t)}, \phi_t)$ ▷ lifting symbolic policy into the blended space
  $\pi^{(t)} \leftarrow \textsc{Update}(\pi^{(t)})$ ▷ policy gradient in neural policy space
  $(\pi_S^{(t+1)}, \phi_{t+1}) \leftarrow \textsc{Project}(\pi^{(t)})$ ▷ synthesize shield and invariant
return Policy $\pi^{(T)}$

The main REVEL procedure is presented in Algorithm 4.1. This algorithm presents a particular instance of mirror descent, in which the neurosymbolic policy class $\mathcal{H}$ consists of neural networks (from the neural class $\mathcal{F}$) wrapped with shields (from the programmatic policy class $\mathcal{G}$). These policies are combined using a monitor in the form of an inductive invariant ϕ. Formally, given a neural network $\pi_N \in \mathcal{F}$, a shield $\pi_S \in \mathcal{G}$, and an invariant ϕ which proves $\text{Verf}(\pi_S)$, we construct a neurosymbolic policy:

$$\pi(\boldsymbol{s}) = \text{if } P^{\#}(\boldsymbol{s}, \pi_N(\boldsymbol{s})) \subseteq \phi \text{ then } \pi_N(\boldsymbol{s}) \text{ else } \pi_S(s).$$

We will use the notation $\pi = (\pi_S, \phi, \pi_N)$ to denote the above neurosymbolic policy.

The "true" branch of this definition represents the normal mode of the policy. That is, we prefer to use the neural policy π_N whenever possible because this policy generally has higher performance than π_S. The conditional

$P^{\#}(\boldsymbol{s}, \pi_N(\boldsymbol{s})) \subseteq \phi$ ensures the safety of the overall neurosymbolic policy using the same invariant which proves the safety of the shield. If this condition holds, then the action $\pi_N(\boldsymbol{s})$ is guaranteed to be safe because it can only ever lead to states which are in ϕ (and ϕ does not overlap with the unsafe states). If the condition does not hold, there may be a transition from state $\boldsymbol{s}$ under action $\pi_N(\boldsymbol{s})$ which leads to a state outside of ϕ. At that point, we can no longer be sure the policy will behave safely. Therefore, if the condition does not hold, we use the shield π_S rather than the network. Because ϕ is an invariant of the shield, we know that $P^{\#}(\boldsymbol{s}, \pi_S(\boldsymbol{s}))$ is always going to be in ϕ. Therefore, for either value of the conditional, the overall neurosymbolic policy is safe.

Algorithm 4.1 starts with a (manually constructed) initial shield $\pi_S^{(0)} \in \mathcal{G}$ and a corresponding invariant ϕ_0, then iteratively applies the three steps of mirror descent:

1. LIFT takes a shield $\pi_S \in \mathcal{G}$ along with an invariant ϕ and constructs a policy $(\pi_S, \phi, \pi_S) \in \mathcal{H}$. The neural component of this policy is just the initial shield, but represented as a neural network. In practice, this can be constructed by training a randomly initialized neural network to imitate π_S using any appropriate imitation learning algorithm (e.g., DAGGER). Because the safety of the mixed policy only depends on π_S and ϕ, this lifted policy is still safe even if the imitation learning algorithm doesn't converge perfectly.

2. UPDATE performs a series of approximate gradient updates to the neu-

rosymbolic policy π. Specifically, because gradients are not generally available for the programmatic policy class $\mathcal{G}$, we approximate gradients on the neurosymbolic policy by applying gradient updates in the neural component only. That is, after a gradient update, the policy (π_S, ϕ, π_N) becomes $(\pi_S, \phi, \pi_N + \eta \nabla_{\mathcal{F}} R(\pi))$ for a learning rate η. Because the update step does not change π_S or ϕ, safety is always maintained in this step.

3. PROJECT takes a neurosymbolic policy $\pi = (\pi_S, \phi, \pi_N)$ and finds the nearest symbolic policy

$$\pi'_S = \underset{\pi''_S \in \mathcal{G}, \text{Verf}(\pi''_S)}{\arg\min} D(\pi''_S, \pi)$$

where D is a predetermined Bregman divergence. Along with π'_S, we also compute a new invariant ϕ' which is used to prove $\text{Verf}(\pi'_S)$. The exact algorithm used for this projection depends on the chosen class of symbolic shields. One instantiation (used in our implementation) is described in Section 4.2.1.

4.2.1 Instantiation with Piecewise Linear Shields

In order to implement REVEL, we must choose a class $\mathcal{G}$ of shields. Policies in $\mathcal{G}$ must be expressive enough to allow high performance, but structured enough to support verification. For this implementation, we use *deterministic,*

piecewise-linear policies of the form:

$$\pi_S(\boldsymbol{s}) = \begin{cases} g_1(\boldsymbol{s}) & \text{if } \chi_1(\boldsymbol{s}) \\ g_2(\boldsymbol{s}) & \text{if } \chi_2(\boldsymbol{s}) \wedge \neg\chi_1(\boldsymbol{s}) \\ \dots & \\ g_n(\boldsymbol{s}) & \text{if } \chi_n(\boldsymbol{s}) \wedge \left(\bigwedge_{i=1}^{n-1} \neg\chi_i(\boldsymbol{s})\right) \end{cases}$$

where $\chi_1, \dots, \chi_n$ are linear predicates partition the state space and each g_i is an affine function. For brevity, we will write the shield policy π_S as a list of pairs (g_i, χ_i). We refer to the part of the state space defined by $\chi_i \wedge \bigwedge_{j=1}^{i-1} \neg\chi_j$ as the *region* for the linear function g_i and denote it by $\text{Region}(g_i)$.

Now we discuss the implementation of Algotirhm 4.1. Becuse the LIFT and UPDATE operations are agnostic to the choice of shield, we will only focus on PROJECT. Recall that the goal of this projection is to find a shield π_S which minimizes the imitation distances $D(\pi_S, \pi)$ for a given neurosymbolic policy π.

Algorithm 4.2 Implementation of projection

Input: A policy $\pi = (\pi_S, \phi, \pi_N)$ where $\pi_S = [(g_1, \chi_1), \dots, (g_n, \chi_n)]$
$\pi_S^* \leftarrow \pi_S$
for $t = 1, \dots, T$ **do**
  $\psi \leftarrow \text{CUTTINGPLANE}(\chi_i)$ for heuristically selected i
  $g_i^1 \leftarrow \text{IMITATESAFELY}(\pi_N, g_i, \chi_i \wedge \psi)$
  $g_i^2 \leftarrow \text{IMITATESAFELY}(\pi_N, g_i, \chi_i \wedge \neg\psi)$
  $\pi'_S \leftarrow \text{SPLIT}(\pi_S, i, (g_i^1, \chi_i \wedge \psi), (g_i^2, \chi_i \wedge \neg\psi))$
  if $D(\pi'_S, \pi) < D(\pi_S^*, \pi)$ **then**
    $\pi_S^* \leftarrow \pi'_S$
$\phi^* \leftarrow \text{SAFESPACE}(\pi_S^*)$
return (π_S^*, ϕ^*)

The projection operation is described in Algorithm 4.2. We start with

an initial neurosymbolic policy $\pi = (\pi_S, \phi, \pi_N)$ and iteratively refine the shield π_S. In each iteration of the algorithm, we heuristically identify one component g_i with region χ_i, then perform the following steps:

(i) Sample a *cuttting plane* which creates a more fine-grained partitioning of the state space by splitting the region χ_i into two new regions χ_i^1 and χ_i^2.

(ii) For each new region χ_i^j, call ImitateSafely to construct a safe linear policy g_i^j along with a corresponding invariant. The new policy minimizes $D(g_i^j, \pi)$ within the region χ_i^j.

(iii) Replace (g_i, χ_i) with the two new components, resulting in a new, refined shield π_S'.

After repeating this process for a fixed number of iterations, we return the the most optimal shield which was constructed during the procedure, along with a corresponding invariant.

Algorithm 4.3 Safely Imitating a policy using a given starting point and partition

Input: A neural policy π_N, a region χ, and a linear policy g.
$g^* \leftarrow g$.
while $*$ **do**
  $S \leftarrow$ ComputeSafeRegion(g^*, χ)
  $g^* \leftarrow g^* - \alpha \nabla D(g^*, \pi_N)$
  $g^* \leftarrow \text{proj}_S\, g^*$
return g^*

Algorithm 4.2 relies on a subroutine ImitateSafely in order to find the *safe* linear policy which best imitates a given neurosymbolic policy within a particular region of the state space. This procedure is further detailed in Algorithm 4.3. In each iteration of this algorithm, we first compute a safe hyperinterval S in the parameter space of g^* over the region χ. In order to compute this safe region, we start with with a region which contains every possible gradient step from g^*. This region might contain some unsafe policies, so we then search for unsafe controllers in the region S. When an unsafe controller is found, we trim S in order to remove that controller. This trimming process continues until the region S can be verified using abstract interpretation. Once the region has been computed, we take a gradient step according to the imitation loss D, which is computed using Dagger in order to gather data for a supervised learning algorithm. Finally, the resulting g^* after the gradient step is projected back into the safe hyperinterval S. Since the final policy g^* is projected into the provably safe region S, we can be sure that the policy returned by ImitateSafely is safe on χ.

Intuitively, recomputing the safe region S at each iteration in ImitateSafely allows the controller to move farther fromt eh starting point g than would otherwise be possible. This is because it is intractable to compute the entire set of safe policies in advance, so our safe region computation can only contain a relatively small subset of the safe policies. However, by recomputing the safe region in each step we effectively "move" the safe region together with the current value of the policy. That is, we only need to verify a thin tube of

policies which surrounds the actual trajectory taken by the gradient descent process through the policy space. For example, if we can verify a region which is at least as large as one gradient step at a particular time step, then the gradient descent process becomes unconstrained for that time step even though we cannot verify the entire state space at once.

4.3 Theoretical Analysis

Compared to standard mirror descent, REVEL introduces two new sources of error. First, because gradients are not available for the neurosymbolic policy class, we approximate the true gradient $\nabla_{\mathcal{H}}$ by the gradient on the neural component $\nabla_{\mathcal{F}}$. Second, the projection step can be inexact due to the abstraction used to compute the safe region to project into. Prior work [99] has studied methods for implementing the projection step with bounded error. Here, we bound the bias in the gradient approximation under some simplifying assumptions and use this result ot prove a regret bound in the final shield that our method converges on. We define a safety indicator Z which is zero whenver the shield is invoked and one otherwise. In order to more closely align this analysis with prior work, we will present the theory in terms of a *cost function* we want to *minimize* as opposed to a reward we want to maximize. We will denote this cost as $J(\pi) = -R(\pi)$. Then we assume:

1. $\mathcal{H}$ is a finite-dimensional vector space equipped with an inner product $\langle \cdot, \cdot \rangle$ and an induced norm $\|\pi\| = \sqrt{\langle \pi, \pi \rangle}$;

2. J is convex in $\mathcal{H}$ and ∇J is L_J-Lipschitz continuous on $\mathcal{H}$;

3. $\mathcal{H}$ is bounded (i.e., $\sup\{\|\pi - \pi'\| \mid \pi, \pi' \in \mathcal{H}\} < \infty$);

4. $\mathbb{E}[1 - Z] \leq \zeta$, i.e., the probability that the shield is invoked at any particular time step is bounded above by ζ;

5. the bias introduced in the sampling process is bounded by β, i.e.,

$$\left\| \mathbb{E}[\hat{\nabla}_{\mathcal{F}} \mid \pi] - \nabla_{\mathcal{F}} J(\pi) \right\| \leq \beta$$

   where $\hat{\nabla}_{\mathcal{F}}$ is the estimated gradient; and

6. for $\boldsymbol{s} \in \mathcal{S}$, $\boldsymbol{a} \in \mathcal{A}$, and $\pi \in \mathcal{H}$, if $\pi(\boldsymbol{a} \mid \boldsymbol{s}) > 0$ then $\pi(\boldsymbol{a} \mid \boldsymbol{s}) > \delta$ for some fixed $\delta > 0$.

The last assumption requries some explanation: intuitively, it amounts to cutting off the tails of the policy distribution so that no action can be possible but arbitrarily unlikely. This is necessary to bound the gradient of the policy.

We will need several standard notions from functional analysis:

Definition 4.1. A differentiable function R is *α-strongly convex* w.r.t. a norm $\|\cdot\|$ if $R(y) \geq R(x) + \langle \nabla R(x), y - x \rangle + \frac{\alpha}{2} \|y - x\|^2$.

Definition 4.2. A differentiable function R is *L_R-strongly smooth* w.r.t. a norm $\|\cdot\|$ if $\|\nabla R(x) - \nabla R(y)\|_* \leq L_R \|x - y\|$.

Definition 4.3. 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. Note that D_R is not necessarily symmetric.

With these preliminaries, we can now examine Algorithm 4.1. The high-level structure of our analysis will be to first prove a bound on the bias induced by approximating $\nabla_{\mathcal{H}}$ with $\nabla_{\mathcal{F}}$. We will then combine this bound with a more general theorem to develop a regret bound on the policies learned with Algorithm 4.1.

For the rest of this section, let R be an α-stronly convex and L_R-smooth functional w.r.t. the norm $\|\cdot\|$ on $\mathcal{H}$. Additionally, let $\nabla_{\mathcal{H}}$ be a Fréchet gradient on $\mathcal{H}$. Then the algorithm can be described as follows: start with $\pi_S^{(0)} \in \mathcal{G}$ (provided by the user) then for each iteration t:

1. Compute a noisy estimate of the gradient $\hat{\nabla} J(\pi_S^{(t-1)}) \approx \nabla J(\pi_S^{(t-1)})$.
2. Update in $\mathcal{H}$: $\nabla R(\pi^{(t)}) = \nabla R(\pi^{(t-1)}) - \eta \hat{\nabla} J(\pi_S^{(t-1)})$.
3. Perform an approximate projection

$$\pi_S^{(t)} = \text{proj}_{\mathcal{G}}^{R}(\pi^{(t)}) \approx \arg\min_{\pi_S \in \mathcal{G}} D_R(\pi_S, \pi^{(t)}).$$

This procedure is approximate functional mirror descent under bandit feedback. We let $D_{\mathcal{G}}$ be the diameter of $\mathcal{G}$, i.e., $D_{\mathcal{G}} = \sup\{\|\pi_S - \pi'_S\| \mid \pi_S, \pi'_S \in \mathcal{G}\}$. Let β and σ^2 be bounds on the bias and variance of the gradient estimate in each iteration. Finally, let ϵ be the bound on the projection error with respect to $\|\cdot\|$. We will make use of the following general theorem:

Theorem 4.1. *(General regret bound for mirror-descent-based RL) [99] Let* $\pi_S^{(1)}, \ldots, \pi_S^{(T)}$ *be a sequence of programmatic policies returned by* REVEL *and*

π_S^* *be the optimal safe programmatic policy. We have the expected regret bound*

$$\mathbb{E}\left[\frac{1}{T}\sum_{i=1}^{T} J\left(\pi_S^{(t)}\right)\right] - J\left(\pi_S^*\right) \leq \frac{L_R D_{\mathcal{G}}^2}{\eta T} + \frac{\epsilon L_R D_{\mathcal{G}}}{\eta} + \frac{\eta\left(\sigma^2 + L_J^2\right)}{\alpha} + \beta D_{\mathcal{G}}.$$

In particular, choosing $\eta = \sqrt{(1/T + \varepsilon)/\sigma^2}$*, this simplifies to*

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

Now we will prove a bound on the bias induced by the gradient approximation:

Lemma 4.2. *(Bound on gradient approximation bias) Let* $D_{\mathcal{H}}$ *be the diameter of* $\mathcal{H}$*, i.e.,* $D = \sup\{\|\pi - \pi'\| \mid \pi, \pi' \in \mathcal{H}\}$ *and let* β_0 *be the sampling bias in the gradient estimation procedure. Then the bias incurred by approximation* $\nabla_{\mathcal{H}} J(\pi)$ *with* $\nabla_{\mathcal{F}} J(\pi)$ *and sampling is bounded by*

$$\left\|\mathbb{E}\left[\hat{\nabla}_{\mathcal{F}} \mid \pi\right] - \nabla_{\mathcal{H}} J(\pi)\right\| = O(\beta_0 + L_J \zeta).$$

Proof. First, we note that

$$\left\|\mathbb{E}\left[\hat{\nabla}_{\mathcal{F}} \mid \pi\right] - \nabla_{\mathcal{H}} J(\pi)\right\| \leq \left\|\mathbb{E}\left[\hat{\nabla}_{\mathcal{F}} \mid \pi\right] - \nabla_{\mathcal{F}} J(\pi)\right\| + \left\|\nabla_{\mathcal{F}} J(\pi) - \nabla_{\mathcal{H}} J(\pi)\right\|.$$

We already assume the first term is bounded by β_0, so we will proceed to bound the second term.

Let $\pi = (\pi_S, \phi, \pi_N)$ be a policy in $\mathcal{H}$. By the policy gradient theorem [94], we have that

$$\nabla_{\mathcal{F}} J(\pi) = \mathbb{E}_{\boldsymbol{s}\sim\rho_\pi, \boldsymbol{a}\sim\pi}\left[\nabla_{\mathcal{F}} \log \pi(\boldsymbol{a} \mid \boldsymbol{s}) Q^\pi(\boldsymbol{s}, \boldsymbol{a})\right] \tag{4.3}$$

where ρ_π is the state distribution induced by π and Q^π is the long-term expected reward from a state $\boldsymbol{s}$ and action $\boldsymbol{a}$. We will omit the distribution subscript in the remainder of the proof for convenience. Now note that if Z is one then $\pi(\boldsymbol{a} \mid \boldsymbol{s}) = \pi_N(\boldsymbol{a} \mid \boldsymbol{s})$, so that in particular

$$\nabla_{\mathcal{F}} \log \pi(\boldsymbol{a} \mid \boldsymbol{s}) Q^\pi(\boldsymbol{s}, \boldsymbol{a}) = \nabla_{\mathcal{H}} \log \pi(\boldsymbol{s} \mid \boldsymbol{a}) Q^\pi(\boldsymbol{s}, \boldsymbol{a}).$$

On the other hand, if Z is zero then $\pi(\boldsymbol{s} \mid \boldsymbol{a})$ is independent of π_N so that

$$\nabla_{\mathcal{F}} \log \pi(\boldsymbol{a} \mid \boldsymbol{s}) Q^\pi(\boldsymbol{s}, \boldsymbol{a}) = 0.$$

Since Z is an indicator and can only take the values zero or one, this means

$$\nabla_{\mathcal{F}} \log \pi(\boldsymbol{a} \mid \boldsymbol{s}) Q^\pi(\boldsymbol{s}, \boldsymbol{a}) = Z \nabla_{\mathcal{H}} \log \pi(\boldsymbol{s} \mid \boldsymbol{a}) Q^\pi(\boldsymbol{s}, \boldsymbol{a})$$

Thus, we can rewrite Equation 4.3 as

$$\begin{aligned}
\nabla_{\mathcal{F}} J(\pi) &= \mathbb{E}\left[Z \nabla_{\mathcal{H}} \log \pi(\boldsymbol{a} \mid \boldsymbol{s}) Q^\pi(\boldsymbol{s}, \boldsymbol{a})\right] \\
&= \mathbb{E}\left[Z\right] \mathbb{E}\left[\nabla_{\mathcal{H}} \log \pi(\boldsymbol{a} \mid \boldsymbol{s}) Q^\pi(\boldsymbol{s}, \boldsymbol{a})\right] + \operatorname{Cov}\left(Z, \nabla_{\mathcal{H}} \log \pi(\boldsymbol{a}, \boldsymbol{s}) Q^\pi(\boldsymbol{s}, \boldsymbol{a})\right) \\
&= \mathbb{E}\left[Z\right] \nabla_{\mathcal{H}} J(\pi) + \operatorname{Cov}\left(Z, \nabla_{\mathcal{H}} \log \pi(\boldsymbol{a} \mid \boldsymbol{s}) Q^\pi(\boldsymbol{s}, \boldsymbol{a})\right) \qquad (4.4)
\end{aligned}$$

Note that the covariance term is a vector where the i'th component is the covariance between Z and the i'th component of the gradient $\nabla^i_{\mathcal{H}}$. Then for each i, by Cuachy-Schwarz we have

$$\left|\operatorname{Cov}\left(Z, \nabla^i_{\mathcal{H}} \log \pi(\boldsymbol{a} \mid \boldsymbol{s}) Q^\pi(\boldsymbol{s}, \boldsymbol{a})\right)\right| \leq \sqrt{\operatorname{Var}\left(Z\right) \operatorname{Var}\left(\nabla^i_{\mathcal{H}} \log \pi(\boldsymbol{a} \mid \boldsymbol{s}) Q^\pi(\boldsymbol{s}, \boldsymbol{a})\right)}.$$

Since $Z \in \{0, 1\}$ we must have $0 \leq \operatorname{Var}(Z) \leq 1$ so that in particular

$$\left|\operatorname{Cov}\left(Z, \nabla^i_{\mathcal{H}} \log \pi(\boldsymbol{a} \mid \boldsymbol{s}) Q^\pi(\boldsymbol{s}, \boldsymbol{a})\right)\right| \leq \sqrt{\operatorname{Var}\left(\nabla^i_{\mathcal{H}} \log \pi(\boldsymbol{a} \mid \boldsymbol{s}) Q^\pi(\boldsymbol{s}, \boldsymbol{a})\right)}.$$

By assumption, for every state-action pair $(\boldsymbol{s}, \boldsymbol{a})$, if $(\boldsymbol{s}, \boldsymbol{a})$ is in the support of ρ_π then $\pi(\boldsymbol{a} \mid \boldsymbol{s}) > \delta$. We also have that $Q^\pi(\boldsymbol{s}, \boldsymbol{a})$ is bounded (because J is Lipschitz on $\mathcal{H}$ and $\mathcal{H}$ is bounded). Then because the gradient of the log is bounded above by one and because $\nabla_{\mathcal{H}}$ is bounded by definition, we have $\|\nabla^i_{\mathcal{H}} \log \pi(\boldsymbol{a} \mid \boldsymbol{s}) Q^\pi(\boldsymbol{s}, \boldsymbol{a})\|$ is bounded. Therefore by Popoviciu's inequality, $\mathrm{Var}(\nabla^i_{\mathcal{H}} \log \pi(\boldsymbol{a} \mid \boldsymbol{s}) Q^\pi(\boldsymbol{s}, \boldsymbol{a}))$ is bounded as well. Choose $B > \mathrm{Var}(\nabla^i_{\mathcal{H}} \log \pi(\boldsymbol{a} \mid \boldsymbol{s}) Q^\pi(\boldsymbol{s}, \boldsymbol{a}))$ for all i. Then we have $\|\mathrm{Var}(\nabla_{\mathcal{H}} \log \pi(\boldsymbol{a} \mid \boldsymbol{s}) Q^\pi(\boldsymbol{s}, \boldsymbol{a}))\|_\infty < \sqrt{B}$, and because $\mathcal{H}$ is finite-dimensional, $\|\mathrm{Var}(\nabla_{\mathcal{H}} \log \pi(\boldsymbol{a} \mid \boldsymbol{s}) Q^\pi(\boldsymbol{s}, \boldsymbol{a}))\| < c\sqrt{B}$ for some constant c for any norm $\|\cdot\|$.

Substituting this into Equation 4.4, we have

$$\|\nabla_{\mathcal{F}} J(\pi) - \mathbb{E}[Z] \nabla_{\mathcal{H}} J(\pi)\| < c\sqrt{B}.$$

Then

$$\begin{aligned}\|\nabla_{\mathcal{F}} J(\pi) - \nabla_{\mathcal{H}} J(\pi)\| &\leq \|\nabla_{\mathcal{F}} J(\pi) - \mathbb{E}[Z] \nabla_{\mathcal{H}} J(\pi)\| + \|\mathbb{E}[Z] \nabla_{\mathcal{H}} J(\pi) - \nabla_{\mathcal{H}} J(\pi)\| \\ &< c\sqrt{B} + \|\mathbb{E}[Z] \nabla_{\mathcal{H}} J(\pi) - \nabla_{\mathcal{H}} J(\pi)\|\end{aligned}$$

By assumption $\nabla_{\mathcal{H}} J(\pi)$ is Lipschitz and $\mathcal{H}$ is bounded. Let $D_{\mathcal{H}}$ be the diameter of $\mathcal{H}$ and recall the L_J is the Lipschitz constant of $\nabla_{\mathcal{H}} J(\pi)$. Choose an arbitrary $\pi^{(0)} \in \mathcal{H}$ and let $J_0 = \nabla_{\mathcal{H}} J(\pi^{(0)})$. Then for any policy $\pi \in \mathcal{H}$ we have $\|\nabla_{\mathcal{H}} J(\pi)\| \leq J_0 + D_{\mathcal{H}} L_J$. Then

$$\begin{aligned}\|\mathbb{E}[Z] \nabla_{\mathcal{H}} J(\pi) - \nabla_{\mathcal{H}} J(\pi)\| &= \|(\mathbb{E}[Z] - 1) \nabla_{\mathcal{H}} J(\pi)\| \\ &= |\mathbb{E}[Z] - 1| \, \|\nabla_{\mathcal{H}} J(\pi)\| \\ &\leq |\mathbb{E}[Z] - 1| \, (J_0 + D_{\mathcal{H}} L_J).\end{aligned}$$

Since Z is an indicator variable, we have $0 \leq \mathbb{E}[Z] \leq 1$ so that $|\mathbb{E}[Z] - 1| = 1 - \mathbb{E}[Z]$. Then finally we assume $D_{\mathcal{H}}$ is a known constant to simplify the presentation and arrive at

$$\|\nabla_{\mathcal{F}} J(\pi) - \nabla_{\mathcal{H}} J(\pi)\| < c\sqrt{B} + (1 - \mathbb{E}[Z])\,(J_0 + D_{\mathcal{H}} L_J) = O(L_J \zeta)$$

and plugging this back into the original triangle inequality we have

$$\left\| \mathbb{E}\left[\hat{\nabla}_{\mathcal{F}} \mid \pi \right] - \nabla_{\mathcal{H}} J(\pi) \right\| = O(\beta_0 + L_J \zeta).$$

□

Now by plugging this bound into Theorem 4.1, we achieve a regret bound on Algorithm 4.1.

Theorem 4.3. *Let $\pi_S^{(1)}, \ldots, \pi_S^{(T)}$ be a sequence of policies in $\mathcal{G}$ returned by* REVEL *and let π_S^* be the optimal programmatic policy. Let β_0 be the sampling bias in the gradient estimation. Choosing a learning rate $\eta = \sqrt{\frac{1}{\sigma^2}\left(\frac{1}{T} + \epsilon\right)}$ we have the expected regret over T iterations:*

$$\mathbb{E}\left[\frac{1}{T} \sum_{i=1}^{T} J\left(\pi_S^{(i)}\right) \right] - J\left(\pi_S^*\right) = O\left(\sigma \sqrt{\frac{1}{T} + \epsilon} + \beta + L_J \zeta \right)$$

Proof. By Lemma 4.2, the overall bound on the bias of the gradient estimate in Algorithm 4.1 is $O(\beta_0 + L_J \zeta)$. Plugging this bound in for β in Theorem 4.1 yields the desired bound. □

This theorem matches the expectation that when a blended policy $\pi = (\pi_S, \phi, \pi_N)$ is allowed to take more actions without the shield intervening (i.e.,

ζ decreases), the regret bound is decreased. Intuitively, this is because when we use the shield, the action we take does not depend on the neural network π_N, so the network does not learn anything useful. However if π is using π_N to choose actions, then we have unbiased gradient information as in standard RL.

4.4 Evaluation

Now we turn to our empirical evaluation of REVEL. Our goal is to answer two questions:

1. How do REVEL policies compare to state-of-the-art RL techniques without formal guarantees of safety? How much safer are they and how much of a performance penalty is incurred?

2. Does REVEL offer benefits compared to prior work in verified exploration which uses static shields [5, 30]?

To answer these questions, we compare REVEL against three baselines: (1) Deep Deterministic Policy Gradients (DDPG) [63], (2) Constrained Policy Optimization (CPO) [3], and (3) a variant of REVEL which never updates the user-provided initial shield. DDPG is a well-known RL algorithm which does not account for safety considerations, so we augment the reward with a penalty for safety violations. CPO is designed for safe exploration, but does not *guarantee* safety.

4.4.1 Benchmarks

We use 10 benchmarks to evaluate the safety and performance each algorithm. These benchmarks include classic control problems, robotics applications, and benchmarks from prior work. For each environment, we hand-constructed a worst-case, piecewise-linear model of the dynamics. These models are based on the pysics of the environment and use non-determinism to approximate nonlinear functions. For example, some of our benchmarks include trigonometric functions which cannot be represented linearly. In these cases, we define piecewise-linear upper and llower bounds to the trigonometric functions. These linear approximations are necessary to make verification feasible. Each benchmark also includes a bounded-time safety property which should hold at all times during training.

The benchmarks are:

1. mountain-car: A continuous version of the classic mountain car problem. In this environment, the goal is to move an underpowered vehicle up a hill (on the right side) by rocking back and forth in a valley to build up momentum. The safety property asserts that the car does not go oer the crest of the hill on the left.

2. road, road-2d, noisy-road, noiys-road-2d: Four variants of an autonomous car control problem. In each case, the car's goal is to move to a specified end position while obeying a given speed limit. The noisy variants introduce noise in the environment, while the 2d variants need to move

through two dimensions to reach the goal (the non-2d variants are one-dimensional).

3. obstacle, ostacle2: A robot moving in 2D space much reach a goal position while avoiding an unsafe region. In obstacle, the unsafe region is placed off to the side so that the agent must avoid it during exploration, but even an unsafe algorithm should converge to safe path. In obstacle2, the unsafe region is between the starting state and the goal, so the policy must learn to move around it.

4. pendulum: A classic pendulum environment. The pendulum starts hanging downward and the system must swing it up so that it is vertical. The safety property in this case is a bound on the angular velocity of the pendulum.

5. acc: An adaptive cruise control benchmark taken from [30] and modified to use a continuous action space. Here the goal is to follow a lead car as closely as possible without crashing into it. At each time step the lead car choose an acceleration at random from a truncated normal distribution.

6. car-racing is similar to obstacle2 except that in this case the agent must reach the goal state and then return to the start, completing one lap around the obstacle.

For each benchmark, we consider a bounded-time variant of the desired safety property. That is, for a fixed T, we guarantee that a policy $\pi = (\pi_S, \phi, \pi_N)$

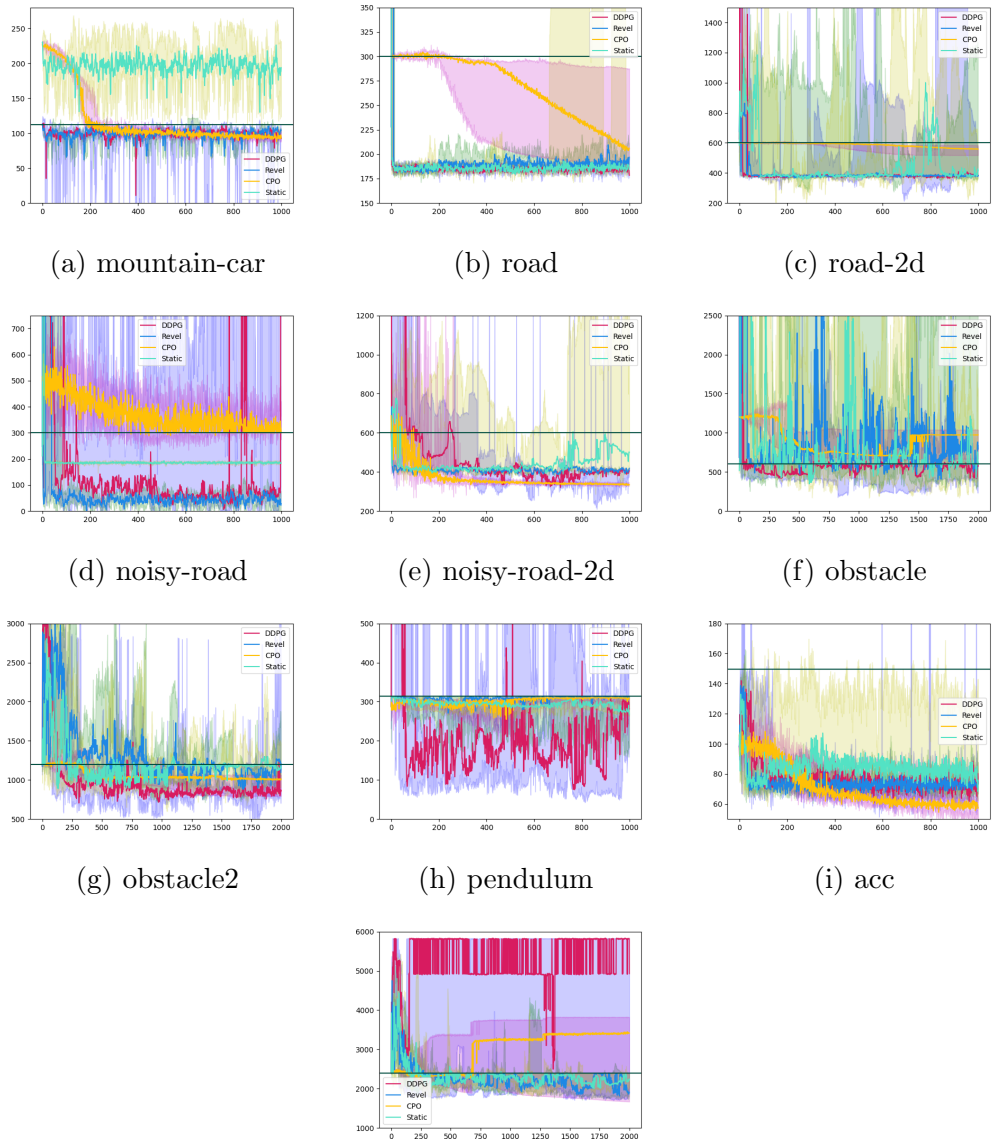
cannot violate the safety property within T time steps starting from any state satisfying ϕ.

4.4.2 Training Details

For most benchmarks, we train for 100,000 environment interactions with a maximum episode length of 100. For mountain-car we use a maximum episode length of 200 and 200,000 total environment interactions. For obstacle, obstacle2, and car-racing we use an episode length of 200 and 400,000 total environment interactions. For every benchmark, we synthesize five new shields at even intervals throughout training. To evaluate CPO we use the implementation provided with the Safety Gym repository [82]. To account for our safety critical benchmarks, we reduce the tolerance for safety violations in this implementation by lowering the corresponding hyperparameter from 25 to 1. For DDPG, we use an implementation from prior work [109], which is also what we base the code for REVEL on. We ran each experiment with five independent, randomly chosen seeds. Note that the chosen number of training episodes was enough for the baselines to converge, in the sense that over the last 25 training episodes we see less than a 2% improvement in policy performance.

4.4.3 Performance

First, we compare the policies learned using REVEL against policies learned using the baselines in terms of the cost incurred by the policies (that



(a) mountain-car (b) road (c) road-2d

(d) noisy-road (e) noisy-road-2d (f) obstacle

(g) obstacle2 (h) pendulum (i) acc

(j) car-racing

Figure 4.1: Training performance comparison between different RL techniques. Note that the y-axis is the cost, so lower is better.

is, the negative of the reward). Figure 4.1 shows the cost over time for each of the policies during training. For each color, the solid line indicates the mean performance while the shaded envelope shows the upper and lower bounds on performance. The horizontal line on each figure represents the performance of the initial shield. The results suggest that:

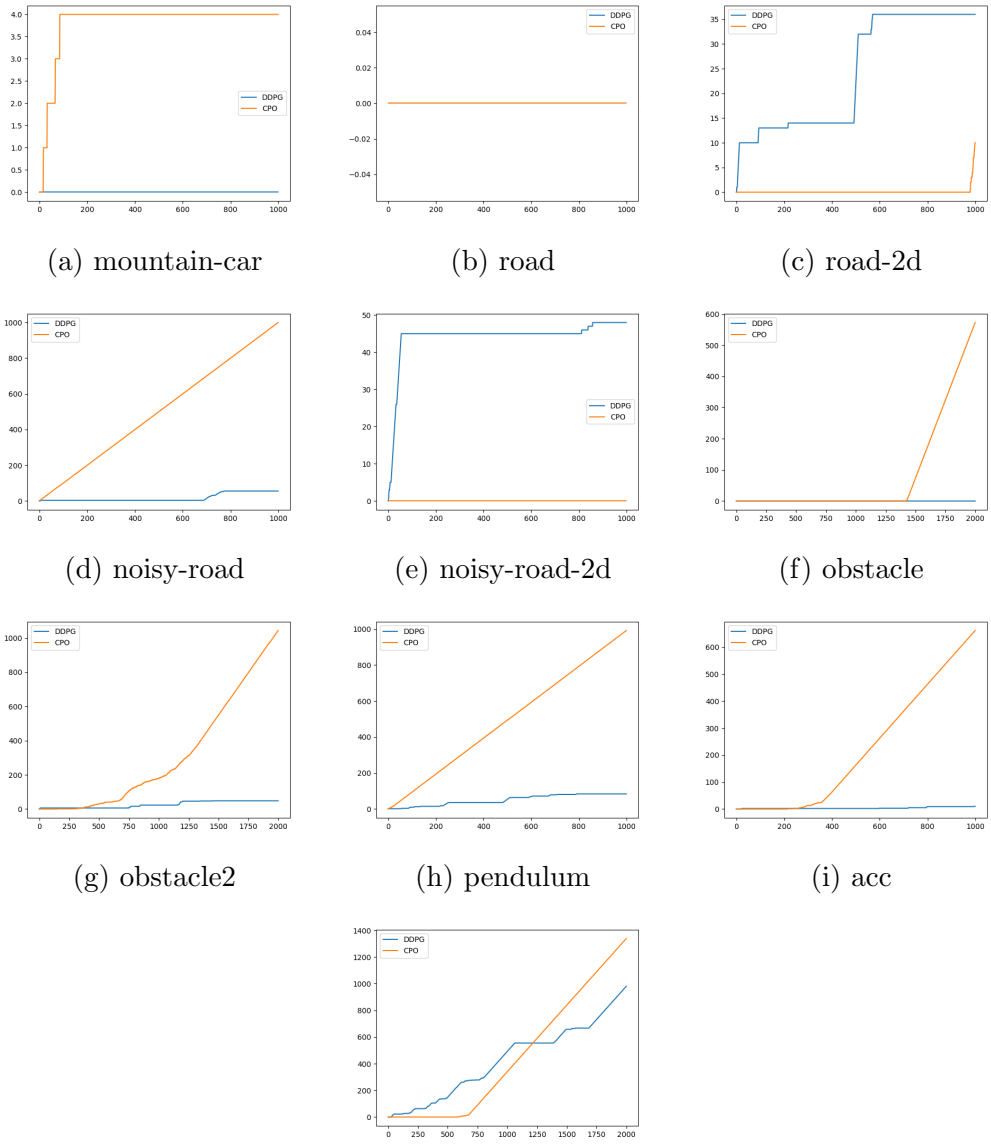
- The performance of REVEL is competitive with (or even better than) DDPG for 7 of the 10 benchmarks. REVEL achieves significantly better reward than DDPG in the "car-racing" benchmark, and the reward is only slightly worse for 2 benchmarks.

- REVEL has better performance than CPO on 4 of the 10 benchmarks and only performs slightly worse on 2. Furthermore, the cost incurred by CPO is significantly worse on 2 benchmarks ("noisy-road" and "car-racing").

- REVEL outperforms the static shielding approach on 4 of the 10 benchmarks. Furthermore, the difference is substantial on 2 of these benchmarks ("noisy-road" and "mountain-car").

4.4.4 Safety

In order to validate whether the safety guarantee provided by REVEL is useful, we consider how DDPG and CPO behave during training. Specifically, Table 4.1 shows the average number of safety violations per run for DDPG and CPO. As we can see from this table, DDPG and CPO both exhibit safety



(a) mountain-car (b) road (c) road-2d

(d) noisy-road (e) noisy-road-2d (f) obstacle

(g) obstacle2 (h) pendulum (i) acc

(j) car-racing

Figure 4.2: Cumulative safety violations during training.

Table 4.1: Safety violations.

Benchmark	DDPG	CPO
mountain-car	0.0	3.6
road	0.0	0.0
road-2d	113.4	70.8
noisy-road	1130.4	8526.4
noisy-road-2d	107.4	0.0
obstacle	12.4	1.0
obstacle2	96.0	118.6
pendulum	92.4	9906.0
acc	4.0	673.0
car-racing	4956.2	22.4

violations in 8 out of the 10 benchmarks. Recall that REVEL policies are proven to have zero safety violations in any of the benchmarks. Figure 4.2 shows how the number of violations caries throughout the training process.

4.4.5 Training cost

REVEL does incur a substantial overhead in terms of computational cost. Table 4.2 shows the time spent in network updates and shield updates for each benchmark, along with the percentage of the total time spent in shield synthesis. The "acc" and "pendulum" benchmarks stand out as having very fast shield updates. For these two benchmarks, the safety properties are relatively simple, so the verificaiton engine is able to come up with the safe shields more quickly. Otherwise, REVEL spends the majority of its time (87% on average) in shield synthesis.

Table 4.2: Training time in seconds for network and shield updates

Benchmark	Network update (s)	Shield update (s)	Shield percentage
mountain-car	1900	5315	73.7%
road	954	9401	90.8%
road-2d	1015	19492	95.1%
noisy-road	962	12793	93.0%
noisy-road-2d	935	25514	96.5%
obstacle	4332	27818	86.5%
obstacle2	4365	21661	83.2%
pendulum	1292	113	8.0%
acc	1097	56	4.9%
car-racing	4361	15892	78.5%

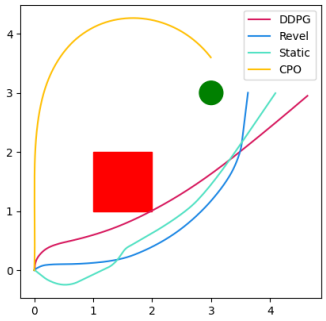


Figure 4.3: Trajectories for obstacle2.

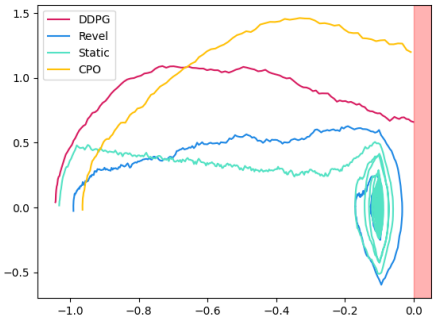


Figure 4.4: Trajectories for acc.

4.4.6 Qualitative Evaluation

In order to provide some intuitive description of the results, we consider some sample trajectories from the trained policies for two of our benchmarks. Figure 4.3 shows the trajectories taken by each of the policies for the obstacle2 benchmark. In this environment, the agent starts in the lower left corner, and the goal is to move to the green circle in the upper right. However the red box

in the middle is unsafe, and the agent must always stay outside of that box. As we can see from Figure 4.3, all of the policies have learned to go around the unsafe region in the center. However DDPG has not reinforced this behavior enough, and due to the lack of safety guarantees, it still enters the corner of the unsafe region. By contrast, the statically shielded policy avoids the unsafe region, but there is a clean bend in the policy where the shield must step in. The policy learned with REVEL avoids the unsafe region while maintaining smoother behavior. In this case, CPO also learns to avoid the unsafe region and reach the goal. (Because the environment is symmetrical, there is no significance to the fact that the CPO curve goes up first and then right as opposed to right first and then up.)

Figure 4.4 shows trajectories for the acc benchmark, which models an adaptive cruise control system. In this environment, the agent is a car whose goal is to follow another car as closely as possible without crashing. The lead car may accelerate or brake at any time with bounded magnitude. In Figure 4.4, the x-axis shows the distance between the controled car and the lead car while the y-axis shows the relatively velocity between the two cars. Here, all of the trajectories start by accelerating to close the gap to the lead car before slowing down again. The statically shielded policy is the first to slow down, thus achieving lower rewards than the other policies. Both the DDPG and CPO policies fail to slow down early enough or quickly enough and they crash into the lead car (represented by the red region on the right of the figure). In contrast, the REVEL policy can more quickly close the gap to

the lead car and is able to slow down later than the statically-shielded policy while still avoiding a crash.

Chapter 5

Safe Exploration in Unknown Environments[1]

While REVEL is able to maintain safety in many cases, it requires a predefined approximation of the environment to do so. In some cases, such environment approximations are not readily available, which makes it impossible to apply REVEL in these environments. In this chapter, we will address this problem by developing a new algorithm which works without a predefined environment approximation.

Our approach, called SPICE [2], is similar to [15] in that we use a learned model to filter out unsafe actions. However, the novel idea in SPICE is to use the symbolic method of *weakest preconditions* [24] to compute, from a single-time-step environment model, a predicate that decides if a given sequence of future actions is safe. Using this predicate, we symbolically compute a *safety shield* [5] that intervenes whenever the current policy proposes an unsafe action. The environment model is repeatedly updated during the learning process using data safely collected using the shield. The computation of the weakest precondition and the shield is repeated, leading to a more refined

[1] This chapter is based on [6]. The author of this dissertation was responsible for the ideas, analysis, and implementation of this technique.

[2] SPICE is available at `https://github.com/gavlegoat/spice`.

shield, on each such update.

The benefit of this approach is sample-efficiency: to construct a safety shield for the next k time steps, SPICE only needs enough data to learn a single-step environment model. We show this benefit using an implementation of the method in which the environment model is given by a piecewise linear function and the shield is computed through quadratic programming (QP). On a suite of challenging continuous control benchmarks from prior work, SPICE has comparable performance as fully neural approaches to safe exploration and incurs far fewer safety violations on average.

This chapter makes the following contributions

- We present the first neurosymbolic framework for safe exploration with learned models of safety.
- We present a theoretical analysis of the safety and performance of our approach.
- We develop an efficient, QP-based instantiation of the approach and show that it offers greater safety than end-to-end neural approaches without a significant performance penalty.

5.1 Background: Weakest Precondition

Our approach to the safe exploration problem is built on *weakest preconditions* [24]. At a high level, weakest preconditions allow us to "translate"

constraints on a program's output to constraints on its input. As a very simple example, consider the function $x \mapsto x + 1$. The weakest precondition for this function with respect to the constraint $ret > 0$ (where ret indicates the return value) would be $x > -1$. In this work, the "program" will be a model of the environment dynamics, with the inputs being state-action pairs and the outputs being states.

For the purposes of this chapter, we present a simplified weakest precondition definition that is tailored towards our setting. Let $f : \mathcal{S} \times \mathcal{A} \to 2^{\mathcal{S}}$ be a nondeterministic transition function. As we will see in Section 5.3, f represents a PAC-style bound on the environment dynamics. We define an alphabet Σ which consists of a set of *symbolic* actions $\omega_0, \ldots, \omega_{H-1}$ and states $\chi_0, \ldots, \chi_H$. Each symbolic state and action can be thought of as a variable representing an *a priori unkonwn* state and action. Let ϕ be a first order formula over Σ. The symbolic states and actions represent a trajectory in the environment defined by f, so they are linked by the relation $\chi_{i+1} \in f(\chi_i, \omega_i)$ for $0 \leq i < H$. Then, for a given i, the weakest precondition of ϕ is a formula ψ over $\Sigma \setminus \{\chi_{i+1}\}$ such that (1) for all $e \in f(\chi_i, \omega_i)$, we have $\psi \implies \phi[\chi_{i+1} \mapsto e]$ and (2) for all ψ' satisfying condition (1), $\psi' \implies \psi$. Here, the notation $\phi[\chi_{i+1} \mapsto e]$ represents the formula ϕ with all instances of χ_{i+1} replaced by the expression e. Intuitively, the first condition ensures that, after taking one environment step from χ_i under action ω_i, the system will always satisfy ϕ, no matter how the nondeterminism of f is resolved. The second condition ensures that ϕ is as permissive as possible, which prevents us from ruling out

states and actions that are safe in reality.

5.2 Symbolic Preconditions in Constrained Exploration

Algorithm 5.1 The main learning algorithm

procedure SPICE
  Initialize an empty dataset D and random policy π
  for epoch in $1 \dots N$ **do**
    if epoch $= 1$ **then**
      $\pi_S \leftarrow \pi$
    else
      $\pi_S \leftarrow \lambda \boldsymbol{s}.\text{WPSHIELD}(M, \boldsymbol{s}, \pi(\boldsymbol{a}))$
    Unroll real trajectores $\{(\boldsymbol{s}_i, \boldsymbol{a}_i, \boldsymbol{s}'_i, r_i)\}$ under π_S
    $D = D \cup \{(\boldsymbol{s}_i, \boldsymbol{a}_i, \boldsymbol{s}'_i, r_i)\}$
    $M \leftarrow \text{LEARNENVMODEL}(D)$
    Optimize π using the simulated environment M

Our approach, Symbolic Preconditions in Constrained Exploration (abbreviated SPICE), uses a learned environment model to both improve sample efficiency and support safety analysis at training time. To do this, we build on top of model-based policy optimization (MBPO) [49]. Similar to MBPO, the model in our approach is used to generate synthetic policy rollout data which can be fed into a model-free learning algorithm to train a policy. In contrast to MBPO, we reuse the environment model to ensure the safety of the system. This dual use of the environment allows for both efficient optimization and safe exploration.

The main training procedure is shown in Algorithm 5.1 and simultaneously learns an environment model M and the policy π. The algorithm

maintains a dataset D of observed environment transitions, which is obtained by executing the current policy π in the environment. SPICE then uses this dataset to learn an environment M, which is used to optimize the current policy π, as done in model-based RL. The key difference of our technique from standard model-based RL is the use of a *shielded policy* π_S when unrolling trajectories to construct dataset D. This is necessary for safe exploration because executing π in the real environment could result in safety violations. In contrast to prior work, the shielded policy π_S in our approach is defined by an online weakest precondition computation which finds a *constraint* over the action space which *symbolically* represents all safe actions. This procedure is described in detail in Section 5.3.

5.3 Shielding with Polyhedral Weakest Preconditions

5.3.1 Overview of Shielding Approach

Algorithm 5.2 Shielding a proposed action

procedure WPSHIELD($M, \boldsymbol{s}_0, \boldsymbol{a}_0^*$)
  $f \leftarrow$ APPROXIMATE($M, \boldsymbol{s}_0, \boldsymbol{a}_0^*$)
  $\phi_H \leftarrow \bigwedge_{i=1}^{H} \chi_i \in \mathcal{S} \setminus \mathcal{S}_U$
  for t from $H-1$ down to 0 **do**
    $\phi_t \leftarrow \text{WP}(\phi_{t+1}, f)$
  $\phi \leftarrow \phi_0[\chi_0 \mapsto \boldsymbol{s}_0]$
  $(\boldsymbol{a}_0, \ldots, \boldsymbol{a}_{H-1}) = \underset{\boldsymbol{a}'_0,\ldots,\boldsymbol{a}'_{H-1} \vDash \phi}{\arg\min} \|\boldsymbol{a}'_0 - \boldsymbol{a}_0^*\|^2$
  return $\boldsymbol{a}_0$

Our high-level online intervention approach is presented in Algorithm 5.2. Given an environment model M, the current state $\boldsymbol{x}_0$ and a proposed action

$\boldsymbol{a}_0^*$, the WpShield procedure chooses a modified action $\boldsymbol{a}_0$ which is as similar as possible to $\boldsymbol{a}_0^*$ while ensuring safety. We consider an action to be *safe* if, after executing that action in the environment, there exists a sequence of follow-up actions $\boldsymbol{a}_1, \ldots, \boldsymbol{a}_{H-1}$ which keeps the system away from the unsafe states over a finite time horizon H. In more detail, our intervention technique works in three steps:

Approximating the environment. Because computing the weakest precondition of a constraint with respect to a complex environment model (e.g., deep neural network) is intractable, Algorithm 5.2 calls the Approximate procedure to obtain a simpler first-order local Taylor approximation to the environment model centered at $(\boldsymbol{s}_0, \boldsymbol{a}_0^*)$. That is, given the environment model M, it computes matrices $\boldsymbol{A}$ and $\boldsymbol{B}$, a vector $\boldsymbol{c}$, and an error ε such that $f(\boldsymbol{s}, \boldsymbol{a}) = \boldsymbol{A}\boldsymbol{s} + \boldsymbol{B}\boldsymbol{a} + \boldsymbol{c} + \Delta$ where Δ is an unknown vector with elements in $[-\varepsilon, \varepsilon]$. The error term is computed based on a normal Taylor series analysis such that with high probability, $M(\boldsymbol{s}, \boldsymbol{a}) \in f(\boldsymbol{s}, \boldsymbol{a})$ in a region close to $\boldsymbol{s}_0$ and $\boldsymbol{a}_0^*$.

Computation of safety constraint. Given a linear approximation f of the environment, Algorithm 5.2 iterates backwards in time, starting with the safety constraint ϕ_H at the end of the time horizon H. In particular, the initial constraint ϕ_H asserts that all (symbolic) states $\chi_1, \ldots, \chi_H$ reached within the time horizon are inside the safe region. Then, the loop inside Algorithm 5.2 uses the wp procedure (described in the next two subsections) to eliminate one symbolic state at a time from the formula ϕ_i. After the loop terminates,

all of the state variables except for χ_0 have been eliminated from ϕ_0, so ϕ_0 is a formula over $\chi_0, \omega_0, \ldots, \omega_{H-1}$. The next line of Algorithm 5.2 simply replaces the symbolic variable χ_0 with the current state $\boldsymbol{s}_0$ in order to find a constraint over only the actions.

Projection onto safe space. The final step of the shielding procedure is to find a sequence $\boldsymbol{a}_0, \ldots, \boldsymbol{a}_{H-1}$ of actions such that (1) ϕ is satisfied and (2) the distance $\|\boldsymbol{a}_0 - \boldsymbol{a}_0^*\|$ is minimized. Here, the first condition enforces the safety of the shielded policy, while the second condition ensures that the shielded policy is as similar as possible to the original one. The notation $\boldsymbol{a}_0, \ldots, \boldsymbol{a}_{H-1} \vDash \phi$ indicates that ϕ is true when the concrete values $\boldsymbol{a}_0, \ldots, \boldsymbol{a}_{H-1}$ are substituted for the symbolic values $\omega_0, \ldots, \omega_{H-1}$ in ϕ. Thus, the $\arg\min$ in Algorithm 5.2 is effectively a projection on the set of action sequences satisfying ϕ. We discuss this optimization problem in Section 5.3.4.

5.3.2 Weakest Preconditions for Polyhedra

In this section, we describe the WP procedure used in Algorithm 5.2 for computing the weakest precondition of a safety constraint ϕ with respect to a linear environment model f. To simplify presentation, we assume that the safe space is given as a convex polyhedron — i.e., all safe states satisfy the linear constraint $\boldsymbol{P}\boldsymbol{s} + \boldsymbol{q} \leq \boldsymbol{0}$. We will show how to relax this restriction in Section 5.3.3.

Recall that our environment approximation f is a linear function with bounded error, so we have constraints over the symbolic states and actions:

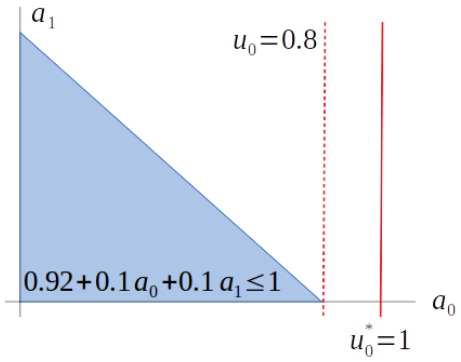


Figure 5.1: Weakest precondition example.

$\chi_{i+1} = \boldsymbol{A}\chi_i + \boldsymbol{B}\omega_i + \boldsymbol{c} + \Delta$ where Δ is an *unknown* vector with elements in $[-\varepsilon, \varepsilon]$. In order to compute the weakest precondition of a linear constraint ϕ with respect to f, we simply replace each instance of χ_{i+1} in ϕ with $\boldsymbol{A}\chi_i + \boldsymbol{B}\omega_i + \boldsymbol{c} + \Delta^*$ where Δ^* is the *most pessimistic* possibility for Δ. Because the safety constraints are linear and the expression for χ_{i+1} is also linear, this substitution results in a new linear formula which is a conjunction of constraints of the form $\boldsymbol{w}^T\nu + \boldsymbol{v}^T\Delta^* \leq y$. For each element Δ_i of Δ, if the coefficient of Δ_i^* is positive in $\boldsymbol{v}$, then we choose $\Delta_i^* = \varepsilon$. Otherwise, we choose $\Delta_i^* = -\varepsilon$. This substitution yields the maximum value of $\boldsymbol{v}^T\Delta^*$ and is therefore the most pessimistic possibility for Δ_i^*.

Example. We illustrate the weakest precondition computation through simple example: Consider a car driving down a (one-dimensional) road whose goal is to reach the other end of the road as quickly as possible while obeying a speed limit. The state of the car is a position x and velocity v. The action space consists of an acceleration a. Assume there is bounded noise in the ve-

locity updates so the dynamics are $x' = x + 0.1v$ and $v' = v + 0.1a + \varepsilon$ where $-0.01 \leq \varepsilon \leq 0.01$ and the safety constraint is $v \leq 1$. Suppose the current velocity is $v_0 = 0.9$ and the safety horizon is two. Then, starting with the safety constraint $v_1 \leq 1 \wedge v_2 \leq 1$ and stepping back through the environment dynamics, we get the precondition $v_1 \leq 1 \wedge v_1 + 0.1a_1 + \varepsilon_1 \leq 1$. Stepping back one more time, we find the condition $v_0 + 0.1a_0 + \varepsilon_2 \leq 1 \wedge v_0 + 0.1a_0 + 0.1a_1 + \varepsilon_1 + \varepsilon_2 \leq 1$. Picking the most pessimistic values for ε_1 and ε_2 to reach $v_0 + 0.1a_0 + 0.01 \leq 1 \wedge v_0 + 0.1a_0 + 0.1a_1 + 0.02 \leq 1$. Since v_0 is specified, we can replace v_0 with 0.9 to simplify this to a constraint over the two actions a_0 and a_1, namely $0.91 + 0.1a_0 \leq 1 \wedge 0.92 + 0.1a_0 + 0.1a_1 \leq 1$. Figure 5.1 shows this region as the shaded triangle on the left. Any pair of actions (a_0, a_1) which lies inside the shaded triangle is guaranteed to satisfy the safety condition for any possible values of ε_1 and ε_2.

5.3.3 Extension to More Complex Safety Constraints

In this section, we extend our weakest precondition computation technique to the setting where the safe region consists of *unions* of convex polyhedra. That is, the state space is represented as a set of matrices $\boldsymbol{P}_i$ and a set of vectors $\boldsymbol{q}_i$ such that $\mathcal{S} \setminus \mathcal{S}_U = \bigcup_{i=1}^{N} \{\boldsymbol{s} \in \mathcal{S} \mid \boldsymbol{P}_i x + \boldsymbol{q}_i \leq \boldsymbol{0}\}$. Note that, while individual polyhedra are limited in their expressive power, unions of polyhedra can approximate reasonable spaces with arbitrary precision. This is because a single polyhedron can approximate a convex set arbitrarily precisely [18], so unions of polyhedra can approximate unions of convex sets.

In this case, the formula ϕ_H in Algorithm 5.2 has the form

$$\phi_H = \bigwedge_{j=1}^{H} \bigvee_{i=1}^{N} \boldsymbol{P}_i \chi_j + \boldsymbol{q}_i \leq \mathbf{0}.$$

However, the weakest precondition of a formula of this kind can be difficult to compute. Because the system may transition between two different polyhedra at each time step, there is a combinatorial explosion in the size of the constraint formula, and a corresponding exponential slowdown in the weakest precondition computation. Therefore, we replace ϕ_H with an approximation

$$\phi'_H = \bigvee_{i=1}^{N} \bigwedge_{j=1}^{H} \boldsymbol{P}_i \chi_j + \boldsymbol{q}_i \leq \mathbf{0}$$

(that is, we swap the conjunction and the disjunction). Note that ϕ_H and ϕ'_H are *not* equivalent, but ϕ'_H is a stronger formula (i.e., $\phi'_H \implies \phi_H$). Thus, any states satisfying ϕ'_H are also guaranteed to satisfy ϕ_H, meaning that they will be safe. More intuitively, this modification asserts that, not only does the state stay within the safe region at each time step, but it stays within *the same polyhedron* at each step within the time horizon.

With this modified formula, we can pull the disjunction outside the weakest precondition, i.e.,

$$\text{WP}\left(\bigvee_{i=1}^{N} \bigwedge_{j=1}^{H} \boldsymbol{P}_i \chi_j + \boldsymbol{q}_i \leq \mathbf{0}, f\right) = \bigvee_{i=1}^{N} \text{WP}\left(\bigwedge_{j=1}^{H} \boldsymbol{P}_i \chi_j + \boldsymbol{q}_i \leq \mathbf{0}, f\right).$$

The conjunctive weakest precondition on the right is of the form described in Section 5.3.2, so this computation can be done efficiently. Moreover, the number of disjuncts does not grow as we iterate through the loop in Algorithm 5.2.

This prevents the weakest precondition formula from growing out of control, allowing for the overall weakest precondition on ϕ'_H to be computed quickly.

Example. Consider an environment which represents a robot moving in 2D space. The state space is four-dimensional, consisting of two position elements x and y and two velocity elements v^x and v^y. The action space consists of two acceleration terms a^x and a^y, giving rise to the dynamics

$$x = x + 0.1v^x \quad y = y + 0.1v^y$$
$$v^x = v^x + 0.1a^x \quad v^y = v^y + 0.1a^y$$

In this environment, the safe space is $x \geq 2 \vee y \leq 1$, so that the upper-left part of the state space is considered unsafe. Choosing a safety horizon of $H = 2$, we start with the initial constraint $(x_1 \geq 2 \vee y_1 \leq 1) \wedge (x_1 \geq 2 \wedge y_2 \leq 1)$. We transform this formula to the stronger formua $(x_1 \geq 2 \wedge x_2 \geq 2) \vee (y_1 \leq 1 \wedge y_2 \leq 1)$. By stepping backwards through the weakest precondition twice, we obtain the following formula over only the current state and future actions:

$$(x_0{+}0.1v_0^x \geq 2 \wedge x_0{+}0.2v_0^x{+}0.01a_0^x \geq 2) \vee (y_0{+}0.1v_0^y \leq 1 \wedge y_0{+}0.2v_0^y{+}0.01a_0^y \leq 1).$$

Intuitively, the approximation we make to the formula ϕ_H does rule out some potentially safe action sequences. This is because it requires the system to stay within *a single* polyhedron over the entire horizon. However, this imprecision can be ameliorated in cases where the different polyhedra comprising the state space overlap one another (and that overlap has non-zero volume). In that case, the overlap between the polyhedra serves as a "transition point",

allowing the system to maintain safety within one polyhedron until it enters the overlap, and then switch to the other polyhedron in order to continue its trajectory.

More formally, we say two polyhedra "overlap" if their intersection has positive volume. That is, polyhedra p_1 and p_2 overlap if $\mu(p_1 \cap p_2) > 0$ where μ is the Lebesgue measure. Often in practical continuous control environments, either this property is satisfied, or it is impossible to verify any safe trajectories at all. This because in continuous control, the system trajectory is a path in the state space, and this path has to move between the different polyhedra defining the safe space. To see how this necessitates our overlapping property, let's take a look at a few possibilities for how the path can pass from one polyhedron p_1 to a second polyhedron p_2. For simplicity, we'll assume the polyhedra are closed, but this argument can be extended straightforwardly to open or partially open polyhedra.

- If the two polyhedra are disconnected, then the system is unable to transition between them because the system trajectory must define a path in the safe region of the state space. Since the two sets are disconnected, the path must pass through the unsafe states, and therefore cannot be safe.

- Suppose the dimension of the state space is n and the intersection of the two polyhedra is an $n-1$ dimensional surface (for example, if the state space is 2D then the polyhedra intersect in a line segment). In this case,

we can add a new polyhedron to the set of safe polyhedra in order to provide an overlap to both p_1 and p_2. Specifically, let X be the set of vertices of $p_1 \cap p_2$. Choose a point x_1 in the interior of p_1 and a point x_2 in the interior of p_2. Now define p' as the convex hull of $X \cup \{x_1, x_2\}$. Note that $p' \subseteq p_1 \cup p_2$, so we can add p' to the set of safe polyhedra without changing the safe state space as a whole. However, p' overlaps with both p_1 and p_2, and therefore the modified environment has the overlapping property.

- Otherwise, $p_1 \cap p_2$ is a lower-dimensional surface. Then for every point $x \in p_1 \cap p_2$ and for every $\epsilon > 0$ there exists an unsafe point x' such that $\|x - x'\| < \epsilon$. In order for the system to transition from p_1 to p_2, it must pass through a point which is arbitrarily close to unsafe. As a result, the system must be arbitrarily fragile — any perturbation can result in unsafe behavior. Because real-world systems are subject to noise and/or modeling error, it would be impossible to be sure the system would be safe in this case.

5.3.4 Projection Onto the Weakest Precondition

After applying the ideas from Section 5.3.3, each piece of the safe space yields a set of linear constraints over the action sequence $\boldsymbol{a}_0, \ldots, \boldsymbol{a}_{H-1}$. That is, ϕ from Algorithm 5.2 has the form

$$\phi = \bigvee_{i=1}^{N} \sum_{j=0}^{H-1} \boldsymbol{G}_{i,j} \boldsymbol{u}_j + \boldsymbol{h}_i \leq 0.$$

Now, we need to find the action sequence satisfying ϕ for which the first action most closely matches the proposed action $\boldsymbol{a}_0^*$. In order to do this, we can minimize the objective function $\|\boldsymbol{a}_0 - \boldsymbol{a}_0^*\|^2$. This function is quadratic, so we can represent this minimization problem as N quadratic programming problems. That is, for each polyhedron $\boldsymbol{P}_i, \boldsymbol{q}_i$ in the safe region, we solve:

$$\begin{aligned} \text{minimize} \quad & \|\boldsymbol{a}_0^* - \boldsymbol{a}_0\|^2 \\ \text{subject to} \quad & \sum_{j=0}^{H-1} \boldsymbol{G}_{i,j}\boldsymbol{a}_j + \boldsymbol{h}_i \leq \boldsymbol{0} \end{aligned}$$

Such problems can be solved efficiently using existing tools. By applying the same technique independently to each piece of the safe state space, we reduce the projection problem to a relatively small number of calls to a quadratic programming solver. This reduction allows the shielding procedure to be applied fast enough to generate the amount of data needed for gradient-based learning.

Example: Consider again Figure 5.1. Suppose the proposed action is $\boldsymbol{a}_0^* = 1$, represented by the solid line in Figure 5.1. Since the proposed action is outside of the safe region, the projection operation will find the point inside the safe region that minimizes the distance *along the* a_0 *axis only.* This leads to the dashed line in Figure 5.1, which is the action $\boldsymbol{a}_0$ that is as close as possible to $\boldsymbol{a}_0^*$ while still intersecting the safe region represented by the shaded triangle. Therefore, in this case, WpShield would return 0.8 as the safe action.

5.4 Theoretical Results

We will now develop theoretical results on the safety and performance of agents trained with SPICE. To do so, we will need to make several assumptions:

Assumption 5.1. The function APPROXIMATE returns a sound, nondeterministic approximation of M in a region reachable from state $\boldsymbol{s}_0$ over a time horizon H. That is, let $\mathcal{S}_R$ be the set of all states $\boldsymbol{s}$ for which there exists a sequence of actions under which the system can transition from $\boldsymbol{s}_0$ to $\boldsymbol{s}$ within H time steps. Then if $f = \textsc{Approximate}(M, \boldsymbol{s}_0, \boldsymbol{a}_0^*)$ then for all $\boldsymbol{s} \in \mathcal{S}_R$ and $\boldsymbol{a} \in \mathcal{A}$, $M(\boldsymbol{s}, \boldsymbol{a}) \in f(\boldsymbol{s}, \boldsymbol{a})$.

Assumption 5.2. The model learning procedure returns a model which is close to the actual environment with high probability. That is, if M is a learned environment model then for all $\boldsymbol{s}, \boldsymbol{a}$,

$$\mathbb{P}_{\boldsymbol{s}' \sim P(\cdot|\boldsymbol{s},\boldsymbol{a})} \left[\|M(\boldsymbol{s}, \boldsymbol{a}) - \boldsymbol{s}'\| > \varepsilon \right] < \delta.$$

Definition 5.1. A state $\boldsymbol{s}_0$ is said to have *realizable safety* over a time horizon H if there exists a sequence of actions $\boldsymbol{a}, \ldots, \boldsymbol{a}_{H-1}$ such that, when $\boldsymbol{s}_0, \ldots, \boldsymbol{s}_H$ is the trajectory unrolled starting from $\boldsymbol{s}_0$ in the true environment, the formula ϕ inside $\textsc{WpShield}(M, \boldsymbol{s}_i, \pi(\boldsymbol{s}_i))$ is satisfiable for all i.

Definition 5.1 formalizes the idea that there exists an intervention at a particular state which can steer the system to safety. If no intervention exists, then of course we are not able to guarantee safety.

Lemma 5.1. *(*WpShield *is safe under bounded error.) Let H be a time horizon, $\boldsymbol{s}_0$ be a state with realizable safety over H, M be an environment model, and π be a policy. Choose ε such that for all states $\boldsymbol{s}$ and actions $\boldsymbol{a}$, $\|M(\boldsymbol{s}, \boldsymbol{a}) - \boldsymbol{s}'\| \leq \varepsilon$ where $\boldsymbol{s}'$ is sampled from the true environment transition at $\boldsymbol{s}$ and $\boldsymbol{a}$. For $0 \leq i < H$ let $\boldsymbol{a}_i = \text{WpShield}(M, \boldsymbol{s}_i, \pi(\boldsymbol{s}_i))$ and let $\boldsymbol{s}_{i+1}$ be sampled from the true environment at $\boldsymbol{s}_i$ and $\boldsymbol{a}_i$. Then for $0 \leq i \leq H$, $\boldsymbol{s}_i \notin \mathcal{S}_U$.*

Proof. Combining Assumption 5.1 with condition 1 in the definition of weakest preconditions, we conclude that for all $e \in M(\boldsymbol{s}_i, \boldsymbol{a}_i)$, $\text{wp}(\phi_{i+1}, f) \implies \phi_{i+1}[\boldsymbol{s}_{i+1} \mapsto e]$. Stepping backward through the loop in WpShield, we find that for all $e_i \in M(\boldsymbol{s}_{i-1}, \boldsymbol{a}_{i-1})$ for $1 \leq i \leq H$, $\phi_0 \implies \phi_H[\boldsymbol{s}_1 \mapsto e_1, \ldots, \boldsymbol{s}_H \mapsto e_H]$. Because ϕ_H asserts the safety of the system, we have that ϕ_0 also implies that the system is safe. Then because the actions returned by WpShield are constrained to satisfy ϕ_0, we also have that $\boldsymbol{s}_i$ for $0 \leq i \leq H$ are safe. □

Theorem 5.2. *Let $\boldsymbol{s}_0$ be a state with realizable safety and let π be any policy. For $0 \leq i < H$, let $\boldsymbol{a}_i = \text{WpShield}(M, \boldsymbol{s}_i, \pi(\boldsymbol{s}_i))$ and let $\boldsymbol{s}_{i+1}$ be the result of taking action $\boldsymbol{a}_i$ at state $\boldsymbol{s}_i$. Then with probability at least $(1 - \delta_M)^i$, $\boldsymbol{s}_i$ is safe.*

Proof. By Lemma 5.1, if $\|M(\boldsymbol{x}, \boldsymbol{u}) - \boldsymbol{x}'\| \leq \varepsilon$ then $\boldsymbol{x}_i$ is safe for all $1 \leq i \leq H$. By Assumption 5.2, $\|M(\boldsymbol{x}, \boldsymbol{u}) - \boldsymbol{x}'\| \leq \varepsilon$ with probability at least $1 - \delta$. Then at each time step, with probability at most δ the assumption of Lemma 5.1 is

violated. Therefore after i time steps, the probability that Lemma 5.1 can be applied is at least $(1-\delta)^i$, so $\boldsymbol{x}_i$ is safe with probability at least $(1-\delta)^i$. $\square$

This theorem shows why SPICE is better able to maintain safety compared to prior work. Intuitively, constraint violations can only occur in SPICE when the environment model is incorrect. In contrast, statistical approaches to safe exploration are subject to safety violations caused by *either* modeling error *or* actions which are not safe even with respect to the environment model. Note that for a safety level δ and horizon H, a modeling error can be computed as $\delta_M < 1 - (1-\delta)/\exp(H-1)$.

The performance analysis is based on treating Algorithm 5.1 as a functional mirror descent in the policy space, similar to [99] and [8]. We assume a class of neural policies $\mathcal{F}$, a class of safe policies $\mathcal{G}$, and a joint class $\mathcal{H}$ of neurosymbolic policies. We proceed by considering the shielded policy $\lambda\boldsymbol{x}.\textsc{WpShield}(M, \boldsymbol{x}, \pi_N(\boldsymbol{x}))$ to be a *projection* of the neural policy π_N into $\mathcal{G}$ for a Bregman divergence D_F defined by a function F. We define a safety indicator Z which is one whenever $\textsc{WpShield}(M, \boldsymbol{x}, \pi^{(i)}(\boldsymbol{x})) = \pi^{(i)}(\boldsymbol{x})$ and zero otherwise, and we let $\zeta = \mathbb{E}[1-Z]$. We additionall assume:

1. $\mathcal{H}$ is a vector space equipped with an inner product $\langle\cdot,\cdot\rangle$ and induced norm $\|\pi\| = \sqrt{\langle\pi,\pi\rangle}$;
2. The long-term reward R is L_R-Lipschitz;
3. F is a convex function on $\mathcal{H}$, and ∇F is L_F-Lipschitz continuous on $\mathcal{H}$;

4. $\mathcal{H}$ is bounded (i.e., $\sup\{\|\pi - \pi'\| \mid \pi, \pi' \in \mathcal{H}\} < \infty$);

5. $\mathbb{E}[1 - Z] \leq \zeta$, i.e., the probability that the shield modifies the action is bounded above by ζ;

6. the bias introduced in the sampling process is bounded by β, i.e., $\|\mathbb{E}[\widehat{\nabla}_{\mathcal{F}} \mid \pi] - \nabla_{\mathcal{F}} R(\pi)\| \leq \beta$, where $\widehat{\nabla}_{\mathcal{F}}$ is the estimated gradient;

7. for $\boldsymbol{s} \in \mathcal{S}$, $\boldsymbol{a} \in \mathcal{A}$, and policy $\pi \in \mathcal{H}$, if $\pi(\boldsymbol{a} \mid \boldsymbol{s}) > 0$ then $\pi(\boldsymbol{a} \mid \boldsymbol{s}) > \delta$ for some fixed $\delta > 0$;

8. the KL-divergence between the true environment dynamics and the model dynamics are is bounded by ϵ_m; and

9. the TV-divergence between the policy used to gather data and the policy being trained is bounded by ϵ_π.

For the following regret bound, we will need three useful lemmas from prior work. These lemmas are reproduced below for completeness.

Lemma 5.3. *([49], Lemma B.3) Let the expected KL-divergence between two transition distributions be bounded by* $\max_t \mathbb{E}_{\boldsymbol{s} \sim p_1^t(\boldsymbol{s})} D_{KL}(p_1(\boldsymbol{s}', \boldsymbol{a} \mid \boldsymbol{s}) \| p_2(\boldsymbol{s}', \boldsymbol{a} \mid \boldsymbol{s})) \leq \epsilon_m$ *and* $\max_{\boldsymbol{s}} D_{TV}(\pi_1(\boldsymbol{a} \mid \boldsymbol{s}) \| \pi_2(\boldsymbol{a} \mid \boldsymbol{s})) < \epsilon_\pi$. *Then the difference in returns under dynamics* p_1 *with policy* π_1 *and* p_2 *with policy* π_2 *is bounded by*

$$|R_{p_1}(\pi_1) - R_{p_2}(\pi_2)| \leq \frac{2 r_{\max} \gamma (\epsilon_\pi + \epsilon_m)}{(1-\gamma)^2} + \frac{2 r_{\max} \epsilon_\pi}{1-\gamma} = O(\epsilon_\pi + \epsilon_m)$$

(where $r_{\max} = \sup_{\boldsymbol{s} \in \mathcal{S}, act \in \mathcal{A}} r(\boldsymbol{s}, \boldsymbol{a})$*).*

Lemma 5.4. *([8], Appendix B) Let D be the diameter of $\mathcal{H}$, i.e., $D = \sup\{\|\pi - \pi'\| \mid \pi, \pi' \in \mathcal{H}\}$. Then the bias incurred by approximating $\nabla_{\mathcal{H}} R(\pi)$ with $\nabla_{\mathcal{F}} r(\pi)$ is bounded by*

$$\left\| \mathbb{E}\left[\hat{\nabla}_{\mathcal{F}} \mid \pi\right] - \nabla_{\mathcal{H}} r(\pi) \right\| = O(\beta + L_R \zeta)$$

Lemma 5.5. *([99], Theorem 4.1) Let $\pi_1, \ldots, \pi_T$ be a sequence of safe policies returned by Algorithm 5.1 (i.e., π_i is the result of calling* WpShield *on the trained policy) and let π^* be the optimal safe policy. Letting β and σ^2 be bounds on the bias and variance of the gradient estimation and let ϵ be a bound on the error incurred due to imprecision in* WpShield*. Then letting $\eta = \sqrt{\frac{1}{\sigma^2}\left(\frac{1}{T} + \epsilon\right)}$, we have the expected regret over T iterations:*

$$R(\pi^*) - \mathbb{E}\left[\frac{1}{T}\sum_{i=1}^{T} R(\pi_i)\right] = O\left(\sigma\sqrt{\frac{1}{T} + \epsilon} + \beta\right).$$

Now using Lemma 5.3, we will bound the gradient bias incurred by using model rollouts rather than true-environment rollouts.

Lemma 5.6. *For a given policy π, the bias in the gradient estimate incurred by using the environment model rather than the true environment is bounded by*

$$\left| \hat{\nabla}_{\mathcal{F}} R(\pi) - \nabla_{\mathcal{H}} R(\pi) \right| = O(\epsilon_m + \epsilon_\pi).$$

Proof. Recall from the policy gradient theorem [94] that

$$\nabla_{\mathcal{F}} R(\pi) = \mathbb{E}_{\boldsymbol{s}\sim\rho_\pi, \boldsymbol{a}\sim\pi}\left[\nabla_{\mathcal{F}} \log \pi(\boldsymbol{a} \mid \boldsymbol{s}) Q^\pi(\boldsymbol{s}, \boldsymbol{a})\right]$$

where ρ_π is the state distribution induced by π and Q^π is the long-term expected reward starting from state $\boldsymbol{s}$ under action $\boldsymbol{a}$. By Lemma 5.3, we have $|Q^\pi(\boldsymbol{s},\boldsymbol{a}) - \hat{Q}^\pi(\boldsymbol{s},\boldsymbol{a})| \leq O(\epsilon_m + \epsilon_\pi)$ where $\hat{Q}^\pi$ is the expected return under the learned environment model. Then because $\log \pi(\boldsymbol{s} \mid \boldsymbol{a})$ is the same regardless of whether we use the environment model or the true environment, we have $\nabla_{\mathcal{F}} \log \pi(\boldsymbol{s} \mid \boldsymbol{a}) = \hat{\nabla}_{\mathcal{F}} \log \pi(\boldsymbol{a} \mid \boldsymbol{s})$ and

$$
\begin{aligned}
&\left|\hat{\nabla}_{\mathcal{F}} R(\pi) - \nabla_{\mathcal{F}} R(\pi)\right| \\
&\qquad = \left|\mathbb{E}\left[\hat{\nabla}_{\mathcal{F}} \log \pi(\boldsymbol{a} \mid \boldsymbol{s}) \hat{Q}^\pi(\boldsymbol{s},\boldsymbol{a})\right] - \mathbb{E}\left[\nabla_{\mathcal{F}} \log \pi(\boldsymbol{a} \mid \boldsymbol{s}) Q^\pi(\boldsymbol{s},\boldsymbol{a})\right]\right| \\
&\qquad = \left|\mathbb{E}\left[\nabla_{\mathcal{F}} \log \pi(\boldsymbol{a} \mid \boldsymbol{s}) \hat{Q}^\pi(\boldsymbol{s},\boldsymbol{a})\right] - \mathbb{E}\left[\nabla_{\mathcal{F}} \log \pi(\boldsymbol{a} \mid \boldsymbol{s}) Q^\pi(\boldsymbol{s},\boldsymbol{a})\right]\right| \\
&\qquad = \left|\mathbb{E}\left[\nabla_{\mathcal{F}} \log \pi(\boldsymbol{a} \mid \boldsymbol{s}) \hat{Q}^\pi(\boldsymbol{s},\boldsymbol{a}) - \nabla_{\mathcal{F}} \log \pi(\boldsymbol{a} \mid \boldsymbol{s}) Q^\pi(\boldsymbol{s},\boldsymbol{a})\right]\right| \\
&\qquad = \left|\mathbb{E}\left[\nabla_{\mathcal{F}} \log \pi(\boldsymbol{a} \mid \boldsymbol{s}) \left(\hat{Q}^\pi(\boldsymbol{s},\boldsymbol{a}) - Q^\pi(\boldsymbol{s},\boldsymbol{a})\right)\right]\right|
\end{aligned}
$$

Now because we assume $\pi(\boldsymbol{a} \mid \boldsymbol{s}) > \delta$ whenever $\pi(\boldsymbol{a} \mid \boldsymbol{s}) > 0$, the gradient of the log is bounded above by a constant. Therefore,

$$
\left|\hat{\nabla}_{\mathcal{F}} R(\pi) - \nabla_{\mathcal{H}} R(\pi)\right| = O(\epsilon_m + \epsilon_\pi).
$$

□

Theorem 5.7. *Let $\pi_S^{(i)}$ for $1 \leq i \leq T$ be a sequence of safe policies learned by* SPICE *(i.e., $\pi_S^{(i)} = \lambda \boldsymbol{s}.\text{WPSHIELD}(M, \boldsymbol{s}, \pi(\boldsymbol{a}))$) and let π_S^* be the optimal safe policy. Additionally Then setting the learning rate $\eta = \sqrt{\frac{1}{\sigma^2}\left(\frac{1}{T} + \epsilon\right)}$, we have the expected regret bound:*

$$
R\left(\pi_S^*\right) - \mathbb{E}\left[\frac{1}{T}\sum_{i=1}^{T} R\left(\pi_S^{(i)}\right)\right] = O\left(\sigma\sqrt{\frac{1}{T} + \epsilon} + \beta + L_R\zeta + \epsilon_m + \epsilon_\pi\right)
$$

Proof. The total bias in gradient estimates is bounded by the sum of (i) the bias incurred by sampling, (ii) the bias incurred by shield interference, and (iii) the bias incurred by using an environment model rather than the true environment. Part (i) is bounded by assumption, part (ii) is bounded by Lemma 5.4, and part (iii) is bounded by Lemma 5.6. Combining these results, we find that the total bias in the gradient estimate is $O(\beta + L_R\zeta + \epsilon_m + \epsilon_\pi)$. Plugging this bound into Lemma 5.5, we reach the desired result. □

This theorem provides a few intuitive results, based on the additive terms in the regret bound. First, ζ is the frequency with which we intervene in network actions and as ζ decreases, the regret bound becomes tighter. This fits our intuition that, as the shield intervenes less and less, we approach standard reinforcement learning. The two terms ϵ_m and ϵ_π are related to how accurately the model captures the true environment dynamics. As the model becomes more accurate, the policy converges to better returns. The other terms are related to standard issues in reinforcement learning, namely the error incurred by using sampling to approximate the gradient.

5.5 Experimental Evaluation

We now turn to a practical evaluation of SPICE.

5.5.1 Implementation and Hyperparameters

Our implementation of SPICE uses PyEarth [84] for model learning and CVXOPT [9] for quadratic programming. Our learning algorithm is based on

MBPO [49] using Soft Actor-Critic [43] as the underlying model-free learning algorithm. Our code is adapted from [97]. We gather real data for 10 episodes for each model update then collect data from 70 simulated episodes before updating the environment model again. We look five time steps into the future during safety analysis. Our SAC implementation uses automatic entropy tuning as proposed in [44]. To compare with CPO we use the original implementation from [3]. Each training process is cut off after 48 hours. We train each benchmark starting from nine distinct seeds.

5.5.2 Benchmarks

We test SPICE using the benchmarks considered in 4, consisting of 10 environments with continuous state and action spaces. The mountain-car and pendulum benchmarks are continuous versions of the corresponding classical control environments. The acc benchmark represents an adaptive cruise control environment. The remaining benchmarks represent various situations arising from robotics. See 4.4.1 for a more complete description of each benchmark.

5.5.3 Baselines

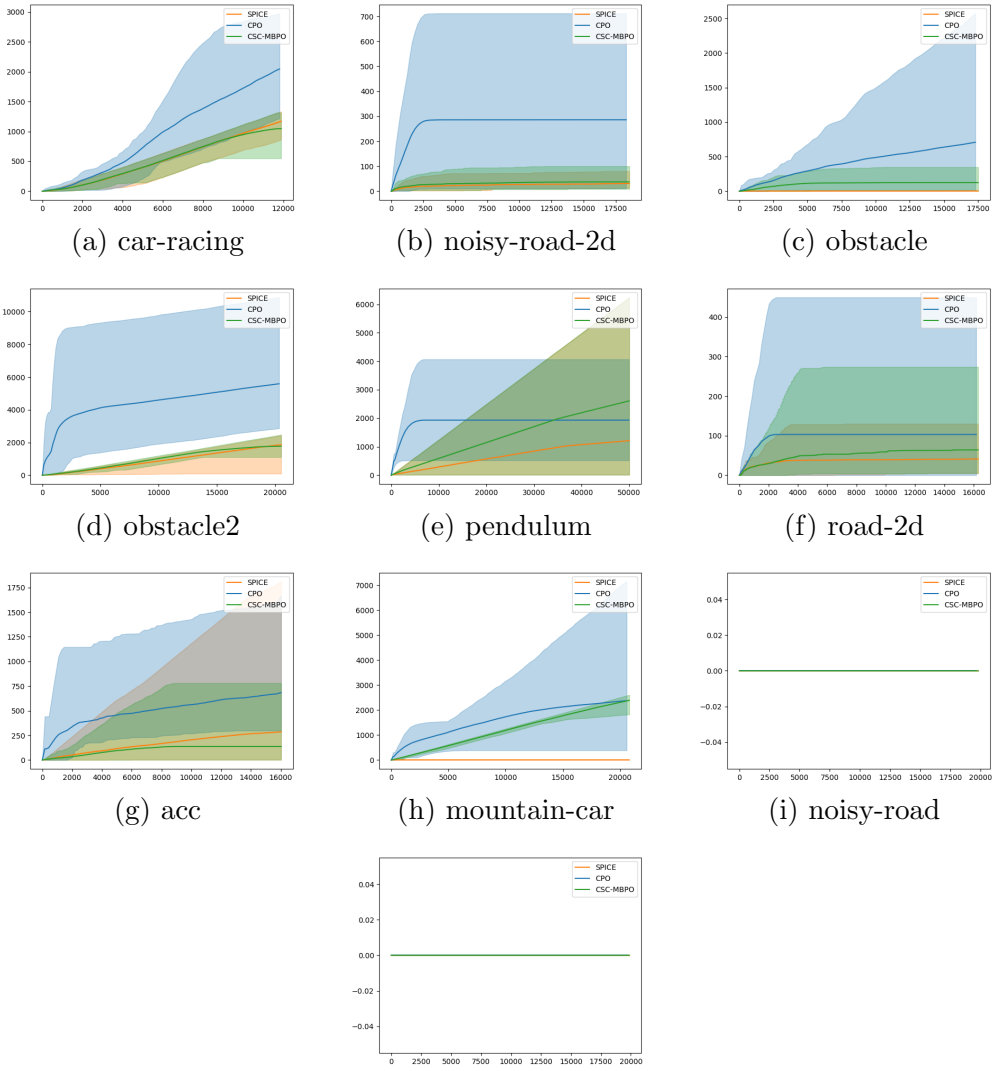
We compare against two baseline approaches: Constrained Policy Optimization (CPO) [3], a model-free safe learning algorithm, and a version of our approach which adopts the conservative safety critic shielding framework from [15] (CSC-MBPO). We additionally tested MPC-RCE [64], another

model-based safe-learning algorithm, but we find that it is too inefficient to be run on our benchmarks. Specifically MPC-RCE was only able to finish on average 162 episodes within a 2-day time period. Therefore, we do not include MPC-RCE in the results presented in this section.

Note that, because the code for [15] is not available, we use a modified version of our code for comparison which we label CSC-MBPO. Our implementation follows Algorithm 5.1 except that WpShield is replaced by an alternative shielding framework. This framework learns a neural safety signal using conservative Q-learning and then resamples actions from the policy until a safe action is chosen, as described in [15]. We chose this implementation in order to give the fairest possible comparison between Spice and the conservative safety critic approach, as the only differences between the two tools in our experiments is the shielding approach. The code for our tool includes our implementation of CSC-MBPO.

5.5.4 Safety

First, we evaluate how well our approach ensures system safety during training. In Table 5.1, we present the number of safety violations encountered during training for our baselines. The last row of the table shows the average increase in the number of safety violations compared to Spice (computed as the geometric mean of the ratio of safety violations for each benchmark). This table shows that Spice is safer than CPO in every benchmark and achieves, on average, a 89% reduction in safety violations. CSC-MBPO is substantially



(a) car-racing (b) noisy-road-2d (c) obstacle

(d) obstacle2 (e) pendulum (f) road-2d

(g) acc (h) mountain-car (i) noisy-road

(j) road

Figure 5.2: Cumulative safety violations over time.

Benchmark	CPO	CSC-MBPO	SPICE
acc	684	137	286
car-racing	2047	1047	1169
mountain-car	2374	2389	6
noisy-road	0	0	0
noisy-road-2d	286	37	31
obstacle	708	124	2
obstacle2	5592	1773	1861
pendulum	1933	2610	1211
road	0	0	0
road-2d	103	64	41
Average	9.48	3.77	1

Table 5.1: Safety violations during training.

safer than CPO, but still not as safe as SPICE. We achieve a 73% reduction in safety violations on average compared to CSC-MBPO. To give a more detailed breakdown, Figure 5.2 shows how the safety violations accumulate over time for several of our benchmarks. The solid line represents the mean over all trials while the shaded envelope shows the minimum and maximum values. As can be seen from these figures, CPO starts to accumulate violations more quickly and continues to violate the safety property more over time than SPICE.

Note that there are a few benchmarks (acc, car-racing, and obstacle2) where SPICE incurs more violations than CSC-MBPO. There are two potential reasons for this increase. First, SPICE relies on choosing a model class through which to compute weakest preconditions (i.e., we need to fix an APPROXIMATE function in Algorithm 5.2). For these experiments, we use a linear approximation, but this can introduce a lot of approximation error. A more complex

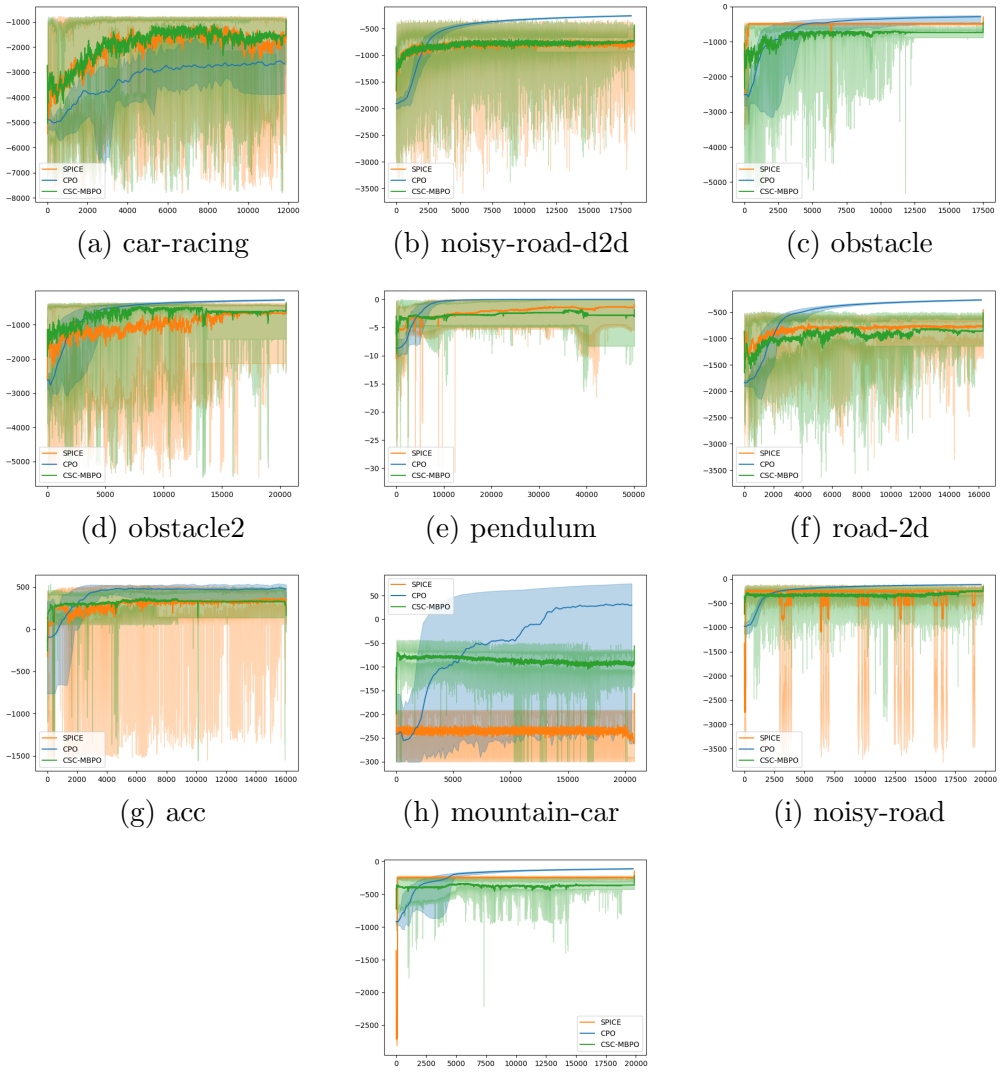
model class allowing a more precise weakest precondition computation may help to reduce safety violations. Second, SPICE uses a bounded-time analysis to determine whether a safety violation can occur within the next few time steps. By contrast, CSC-MBPO uses a neural model to predict the long-term safety of an action. As a result, actions which result in safety violations far into the future may be easier to intercept using the CSC-MBPO approach. Given that SPICE achieves much lower safety violations on average, we think these trade-offs are desirable in many situations.

5.5.5 Performance

We also test the performance of the learned policies on each benchmark in order to understand what impact our safety techniques have on model learning. Figure 5.3 show the average return over time for SPICE and the baselines. These curves show that in most cases SPICE achieves a performance close to that of CPO, and about the same as CSC-MBPO. We believe that the relatively modest performance penalty incurred by SPICE is an acceptable trade-off in some safety-critical systems given the massive improvement in safety.

5.5.6 Qualitative Evaluation

In order to understand how SPICE compares to prior work at a qualitative level, we will examine the trajectories of policies learned by SPICE, CPO, and CSC-MBPO partway through training. In the environment shown in Figure 5.4, the agent controls a robot moving on a 2D plane which must



(a) car-racing (b) noisy-road-d2d (c) obstacle

(d) obstacle2 (e) pendulum (f) road-2d

(g) acc (h) mountain-car (i) noisy-road

(j) road

Figure 5.3: Training curves for SPICE and CPO.

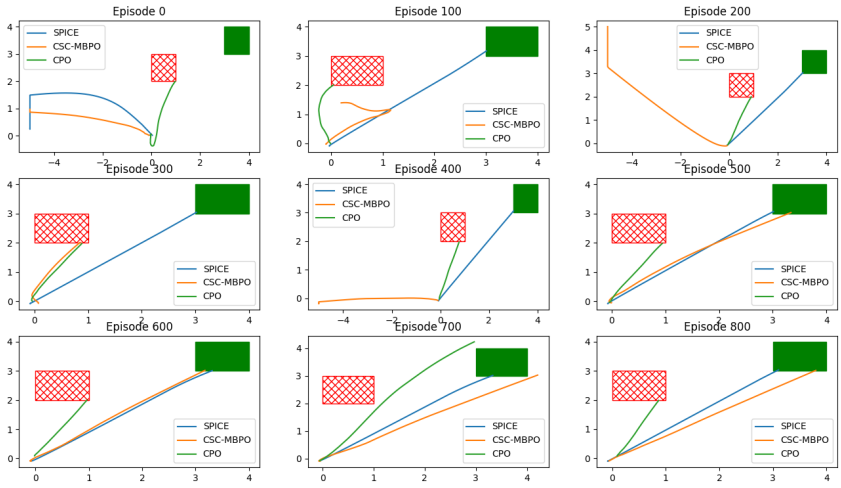


Figure 5.4: Trajectories early in training.

reach the green shaded area while avoiding the red cross-hatched area. For each algorithm, we sampled 100 trajectories and plotted the worst one.

Figure 5.4 shows trajectories sampled from each tool at various stages of training. Specifically, each 100 episodes during training, 100 trajectories were sampled. The plotted trajectories represent the worst samples from this set of 100. The environment represents a robot moving in a 2D plane which must reach the green shaded region while avoiding the red crosshatched region. (Notice that while the two regions appear to move in the figure, the are actually static. The axes in each part of the figure change in order to represent the entirety of the trajectories.) From this visualization, we can see that SPICE is able to quickly find a policy which safely reaches the goal every time. By contrast, CSC-MBPO requires much more training data to find a good policy

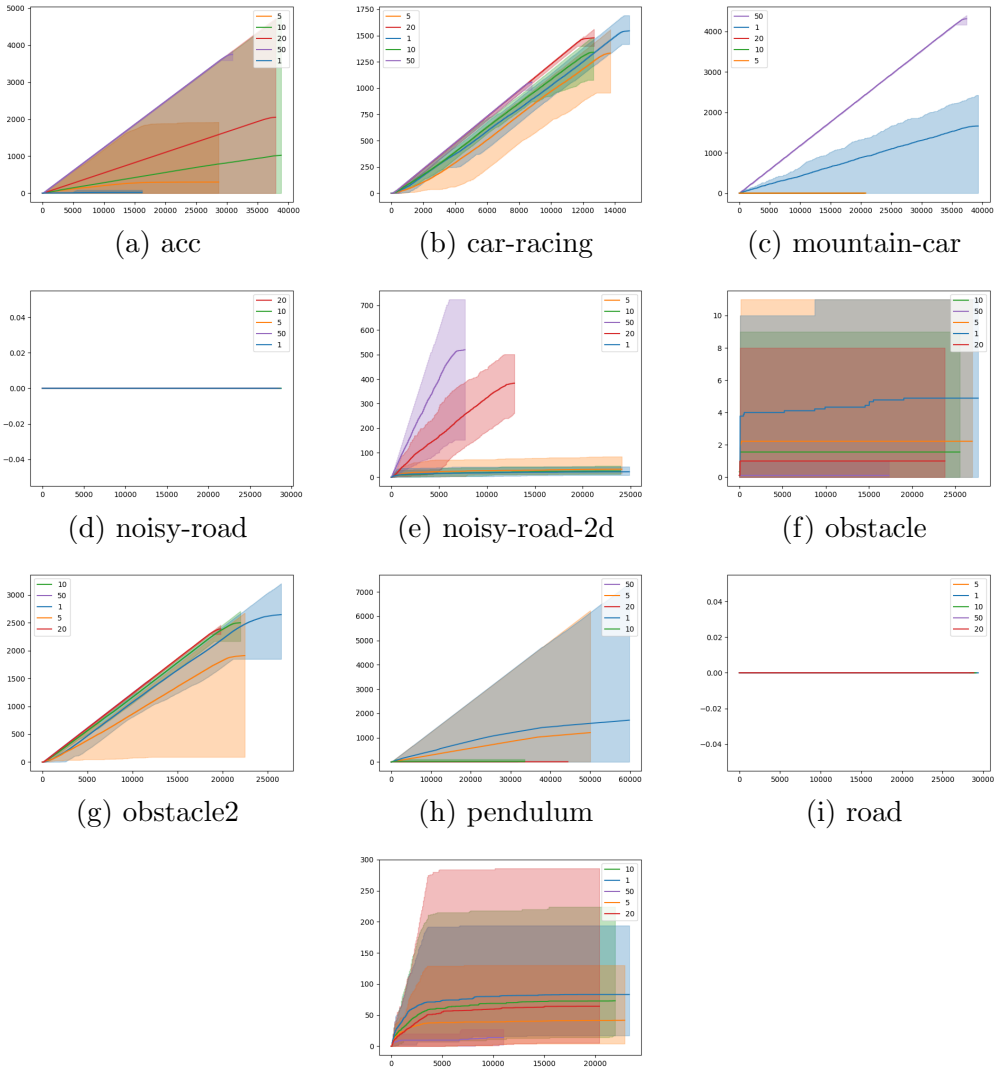
and encounters more safety violations along the way. CPO is also slower to converge and more unsafe than SPICE.

5.5.7 Exploring the Safety Horizon

SPICE relies on choosing a good horizon over which to compute the weakest precondition. We will now explore this tradeoff in more detail. Safety curves for each benchmark under several different choices of horizon are presented in Figure 5.5. The performance curves for each benchmark are shown in Figure 5.6.

There are a few notable phenomena shown in these curves. As expected, in most cases using a safety horizon of one does not give particularly good safety. This is expected because as the safety horizon becomes very small, it is easy for the system to end up in a state where there are no safe actions. The obstacle benchmark shows this trend very clearly: as the safety horizon increases, the number of safety violations decreases.

On the other hand, several benchmarks (e.g., acc, mountain-car, and noisy-road-2d) show a more interesting dynamic: very large safety horizons also lead to an increase in safety violations. This is a little less intuitive because as we look farther into the future, we should be able to avoid more unsafe behaviors. However, in reality there is an explanation for this phenomenon. The imprecision in the environment model (both due to model learning and due to the call to APPROXIMATE) accumulates for each time step we need to look ahead. As a result, large safety horizons lead to a fairly imprecise



(a) acc (b) car-racing (c) mountain-car

(d) noisy-road (e) noisy-road-2d (f) obstacle

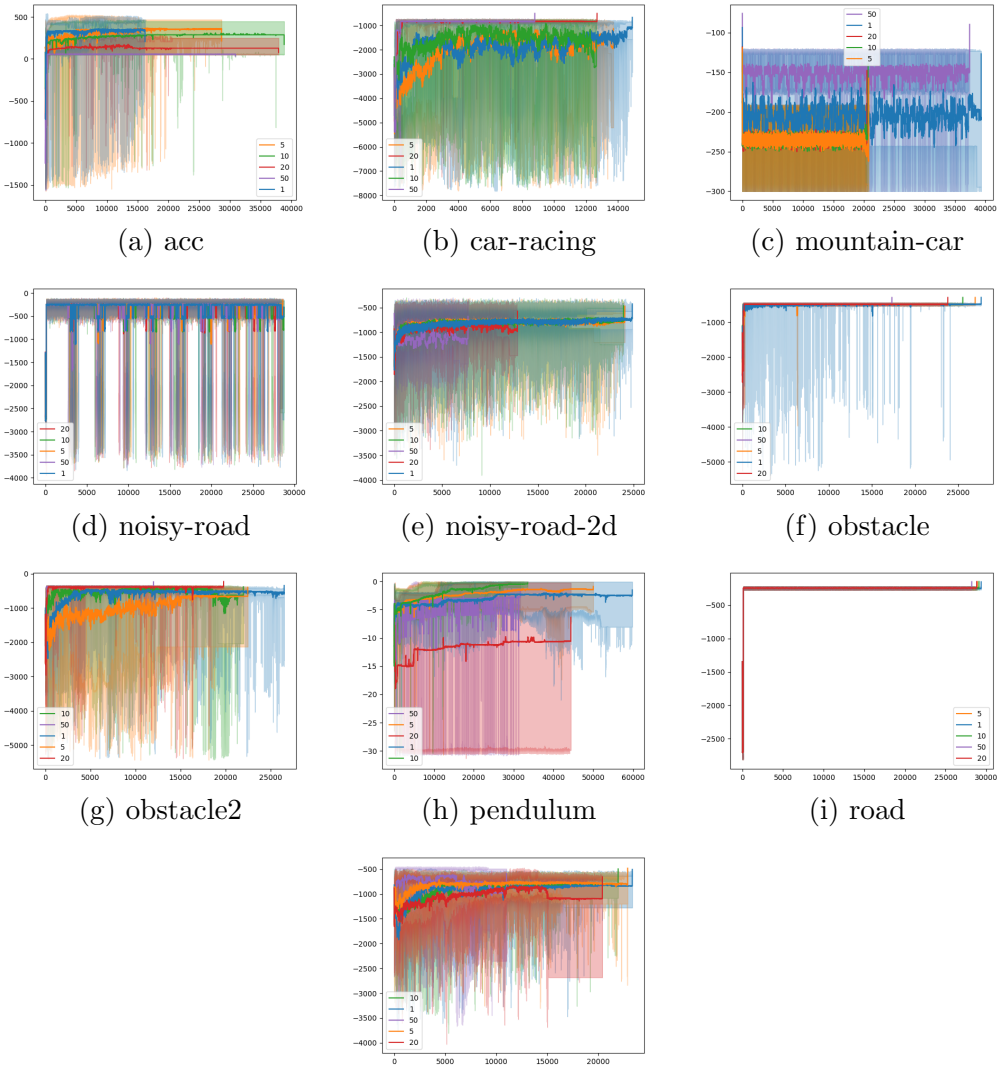
(g) obstacle2 (h) pendulum (i) road

(j) road-2d

Figure 5.5: Safety curves for SPICE using different safety horizons

analysis. Not only does this interfere with exploration, but it can also lead to an infeasible constraint set in the shield. (That is, ϕ in Algorithm 5.2 becomes unsatisfiable.) In this case, the projection in Algorithm 5.2 is ill-defined, so SPICE relies on a simplistic backup controller. This controller is not always able to guarantee safety, leading to an increase in safety violations as the horizon increases.

In practice, we find that a safety horizon of five provides a good amount of safety in most benchmarks without interfering with training. Smaller or larger values can lead to more safety violations while also reducing performance a little in some benchmarks. In general, tuning the safety horizon for each benchmark can yield better results, but for the purposes of this evaluation we have chosen to use the same horizon throughout.



(a) acc (b) car-racing (c) mountain-car

(d) noisy-road (e) noisy-road-2d (f) obstacle

(g) obstacle2 (h) pendulum (i) road

(j) road-2d

Figure 5.6: Training curves for SPICE using different safety horizons

Chapter 6

Scalable Shielding

The approaches described in Chapters 4 and 5 offer strong safety guarantees, but they both suffer from scalability challenges. In both cases, the machine learning algorithms we are building on are able to handle much more complex systems than the programming languages tools we use to ensure safety. In this chapter, we turn our attention to these scalability challenges and develop a new shielding approach which is able to handle complex systems.

The approach presented in this chapter, MSSC, retains the overall shielding structure presented in previous chapters, but implements that shielding framework entirely with neural networks. That is, rather than using abstract interpretation (as in Chapter 4) or weakest preconditions (as in Chapter 5) to develop a backup controller, MSSC uses a gradient-based learning technique. This idea is related to the notion of safety critics proposed in [15], but uses a different shielding structure in order to allow safer shield learning without sacrificing performance. Specifically, while [15] trains a *critic* which can predict unsafe behavior, the only given mechanism for avoiding unsafe behavior is resampling. By contrast, our approach trains the critic along with an explicit backup policy which can provide safe actions. This backup policy

allows for more efficient shielding as well as a more general class of policies.

Because learning a safety critic and backup policy can involve making unsafe decisions, we perform this learning process inside a model of the environment. This allows the agent to explore unsafe behavior in a simulated environment without consequences. In principle, a backup policy trained in an environment model may not enforce safety in the real environment, but we show (empirically) that by using an appropriate model class we can avoid this issue. Once a backup policy has been learned, it can be combined with any other policy in order to enforce safety.

In summary, this chapter makes the following contributions

- We present a novel approach to safe exploration which is able to use complex neural network models for both reward optimization and safety analysis.
- We give preliminary results suggesting that this approach is able to ensure safety more effectively than existing statistcal techniques while scaling to larger systems than existing formal methods-based approaches.

6.1 Background: Safety Critics

Our approach to safe exploration builds on the idea of *safety critics* proposed by Bharadhwaj, et. al. [15]. For a given policy π, a safety critic Q_C evaluates the safety of potential actions under π. That is, for a given state $\boldsymbol{s}$ and action $\boldsymbol{a}$, the value of the critic $Q_C(\boldsymbol{s}, \boldsymbol{a})$ gives the probability that after

taking action $\boldsymbol{a}$ from state $\boldsymbol{s}$, continuing to unroll a trajectory under π results in unsafe behavior.

Safety critics serve two purposes: they act as a monitors to determine whether specific actions may be unsafe and they are used to modify training in order to converge to safe policies. The first use, as monitors, allow for safe exploration by using the critic as a shield. In [15] the shield works by sampling new actions from the same (stochastic) policy until one is deemed safe by the safety critic. This strategy can be computationally inefficient when the policy assigns low probability to safe actions, so in this chapter we will define an improved shield by training an explicit backup policy. The second purpose of safety critics as presented in [15] is to act as a training tool. Specifically, if the safety critic is accurate then the optimal safe policy is the solution to

$$\begin{aligned} \pi^* = &\arg\max_{\pi} R(\pi) \\ &\text{s.t. } \mathbb{E}_{\boldsymbol{s},\boldsymbol{a}\sim\pi}\left[Q_C(\boldsymbol{s},\boldsymbol{a})\right] \leq \delta \end{aligned}$$

Using the method of Lagrange multipliers, this optimization problem may be represented as

$$\pi^* = \arg\max_{\pi}\min_{\lambda\geq 0} R(\pi) - \lambda\left(\mathbb{E}_{\boldsymbol{s},\boldsymbol{a}\sim\pi}\left[Q_C(\boldsymbol{s},\boldsymbol{a})\right] - \delta\right)$$

and solved with alternating gradient descent-ascent.

6.2 Model-Based Shielding with Safety Critics

The key idea of Model-Based Shielding with Safety Critics (MSSC) is to use the real environment to train a high-performance policy π_N while using

a model to train a safe backup policy π_S. Intuitively, learning in the real environment is expected to yield the highest-performing policy because there is no error in the environment behavior. On the other hand, learning in a model is safe because the model is purely virtual and there are no consequences for entering unsafe parts of the state space. (An alternative view is that model-based learning is *sample efficient* with respect to unsafe trajectories, leading to a reduction in safety violations.) The goal of MSSC is to combine the advantages of model-based and model-free learning. The model is used to generate a safe *backup* policy π_S which conforms to safety constraints but may suffer from suboptimal performance. The backup policy is then used as a shield during the training of a high-performance policy π_N in the real environment. This allows us to safely explore policies using the real environment.

In order to create safe combined policies, we adopt the idea of a safety critic from Bharadhwaj, et. al. [15]. Intuitively, Q_C acts as an *inductive invariant* for π_S, and as a result Q_C and π_S can be used together to ensure the safety of arbitrary policies as in Chapter 4. Specifically, for a given policy π_N, we define a shielded policy π as

$$\pi(\boldsymbol{s}) = \text{if } Q_C(\boldsymbol{s}, \pi_N(\boldsymbol{s})) > \delta \text{ then } \pi_S(\boldsymbol{s}) \text{ else } \pi_N(\boldsymbol{s}),$$

(for a given safety level δ) which we denote as $\pi = (\pi_N, Q_C, \pi_S)$. Intuitively, Q_C is used to evaluate the safety of *off-policy* actions (with respect to π_S) and determine whether we may execute those actions, or whether we must use π_S instead.

Algorithm 6.1 Model-Based Shielding with Safety Critics

Input: Environment E, initial critic Q_C, initial policy π_S
Output: Optimal shielded policy
Randomly initialize π_N
Initialize empty dataset $\mathcal{D}$
for N epochs **do**
  Train $\pi_N \leftarrow \arg\max_{\pi'_N} R((\pi'_N, Q_C, \pi_S))$ in E
  Augment $\mathcal{D}$ by rolling out (π_N, Q_C, π_S) in E
  Train a model $M \leftarrow \arg\min_{M'} \mathbb{E}_{(\boldsymbol{s},\boldsymbol{a},\boldsymbol{s}')\sim\mathcal{D}}[\|M'(\boldsymbol{s},\boldsymbol{a}) - \boldsymbol{s}'\|]$
  Jointly train Q_C and π_S in the model M.
return (π_N, Q_C, π_S)

The main MSSC procedure is shown in Algorithm 6.1. It takes as input an initial safety critic and backup policy which can be used to ensure safety in the first iteration of the algorithm[1]. Notice that this initial safety critic may be overly conservative because it will be updated over time to allow better exploration. The backup policy and safety critic are then used to enforce safety in the real environment E while the high-performance policy π_N is trained, as described in Chapter 4. This training can be done with any existing RL algorithm. Once we have trained π_N, we use it in combination with the backup policy and critic to collect data about the environment dynamics and add that data to the dataset $\mathcal{D}$. This dataset is then used to train an environment model.

The final step in the loop of Algorithm 6.1 is to train a new backup policy and safety critic in the learned model M. Formally, the safety critic is

[1]This initial shield can be replaced by an initial dataset which is used to construct an initial model, if obtaining a dataset is easier than constructing a shield.

learned via Q-learning,

$$Q_C{}^{k+1} = \arg\min_{Q_C} \mathbb{E}_{(\boldsymbol{s},\boldsymbol{a},\boldsymbol{s}',c)\sim\mathcal{D}} \left[\left(Q_C(\boldsymbol{s},\boldsymbol{a}) - \hat{\mathcal{B}}_S^{\pi} Q_C{}^{k} \right)^2 \right] \tag{6.1}$$

where c is the immediate cost and $\hat{\mathcal{B}}_S^{\pi}$ is the empirical Bellman operator defined in [56]. The updates to Q_C are made in alternation with updates to the shield policy π_S, which solves

$$\pi_S^{k+1} = \arg\min_{\pi_S} \max_{\lambda \geq 0} \mathbb{E}_{\boldsymbol{s},\boldsymbol{a}\sim\pi_S} \left[D(\pi_S(\boldsymbol{s},\boldsymbol{a}), \pi^k(\boldsymbol{s},\boldsymbol{a})) \right] + \lambda \left(\mathbb{E}_{\boldsymbol{s},\boldsymbol{a}\sim\pi_S} \left[Q_C(\boldsymbol{s},\boldsymbol{a}) \right] - \delta \right) \tag{6.2}$$

for an appropriate distance metric D. As discussed in Section 6.1, Equation 6.2 is a standard Lagrangian approach to constrained optimization. [2] The solution to this saddle point problem is the policy π_S which is as similar as possible to the overall shielded policy π, but which is also safe according to Q_C.

Notice that the solution π_S^{k+1} to Equation 6.2 satisfies the constraint

$$\mathbb{E}_{\boldsymbol{s},\boldsymbol{a}\sim\pi_S} \left[Q_C(\boldsymbol{s},\boldsymbol{a}) \right] \leq \delta.$$

Intuitively, this condition ensures that running π_S^{k+1} is safe with probability at least $1-\delta$. Now in the next iteration of Algorithm 6.1, π_S^{k+1} is combined with π_N to generate $\pi = (\pi_N, Q_C, \pi_S^{k+1})$. The safety of π relies on using Q_C to define an *inductive invariant* of π_S^{k+1}. Formally we define a set of states $\phi = \{\boldsymbol{s} \in \mathcal{S} \mid Q_C(\boldsymbol{s}, \pi_S^{k+1}(\boldsymbol{s})) \leq \delta\}$. Recall that an inductive invariant has to satisfy three properties:

[2] Compared to Section 6.1, we have replaced the policy return with an imitation objective and swapped minima and maxima accordingly

1. the initial states of the system are included in the invariant;

2. the unsafe states are *not* included in the invariant; and

3. for any state $\boldsymbol{s}$ which is included in the invariant, taking one step under π starting from $\boldsymbol{s}$ yields another state which is included in the invariant.

The first condition is trivially satisfied by ϕ as long as there exists a safe policy in the environment. This is because the initial states of the system are always included in the state-marginal distribution induced by any policy. The second condition is also satisfied because if Q_C is accurate then unsafe states will always be assigned a value of one, so they cannot be included in ϕ. The third condition may be analyzed by looking at the structure of $\pi(\boldsymbol{s})$ for some state $\boldsymbol{s} \in \phi$. Because $\boldsymbol{s}$ is included in ϕ we have $Q_C(\boldsymbol{s}, \pi_S^{k+1}(\boldsymbol{s})) \leq \delta$. If $Q_C(\boldsymbol{s}, \pi_N(\boldsymbol{s})) \leq \delta$, then because Q_C is trained on the *undiscounted* cost, we have that for the next state $\boldsymbol{s}'$, $Q_C(\boldsymbol{s}', \pi_S^{k+1}(\boldsymbol{s}')) \leq \delta$ so that $\boldsymbol{s}' \in \phi$. Otherwise, we directly apply the backup policy so we must still have that $\boldsymbol{s}' \in \phi$.

At a theoretical level, Algorithm 6.1 may be viewed as an implementation of REVEL in which the shield consists of a neural backup policy and safety critic. As a result, the analysis presented in Section 4.3 is applicable to Algorithm 6.1. Therefore, under reasonable assumptions, we may apply the regret bound from Theorem 4.3 to MSSC directly.

6.2.1 Shield Generalization

Underlying Algorithm 6.1 is the assumption that a backup policy and safety critic which are trained in a model can be used in the real environment. However it is not immediately clear that this assumption should hold. In prior work on model-based RL, agents have been able to exploit inaccuracies in the environment model to learn policies which have high performance in the model but worse performance in the real environment [21]. There is no reason the same kind of exploitation would not happen with respect to the safety signal in addition to the reward.

Fortunately, because the cost signal is defined analogously to a reward, we can use the same remedies which have been developed for normal model-based RL. Specifically, we adopt the idea of a probabilistic ensemble model from prior work [21, 49]. A probabilistic ensemble consists of B neural networks, each independently randomly initialized and independently trained. For a given state and action, each network outputs the parameters of a probability distribution representing the next state. These parameters capture *aleatoric* uncertainty, i.e., the inherent randomness of the environment. At the same time, the difference between different networks captures *epistemic* uncertainty, i.e., uncertainty in the network parameters arising from incomplete training. This can be understood intuitively by looking at two extreme cases: first, suppose we have no training data. In this case, each network is randomly initialized and there is no relationship between the outputs of each network because nothing is known about the environment. On the other hand,

imagine we have plenty of training data and allow each network to be trained to convergence. Then every network accurately represents the environment, and so for a given state and action, they all output the same parameters. There may still be randomness in the model output, but this randomness is now only due to inherent stochasticity in the environment.

Following [49], we use the probabilistic ensemble model by choosing one network uniformly at random each time the model is called. This alleviates the problem of policies exploiting error in the model because the individual networks in the model have different biases due to their different initializations. We show empirically in Section 6.3 that MSSC is able to apply shields learned in the model to the real environment.

6.3 Evaluation

We now turn our attention to an empirical evaluation of MSSC, using a subset of the benchmarks introduced in Chapter 4 (acc, pendulum, road-2d, and noisy-road-2d). We compare against the same baselines used in Chapter 5.

6.3.1 Implementation and Training Details

We implement MSSC using soft actor-critic [43] as our main learning algorithm for training π_N, using the implementation from [97]. Our environment model is an ensemble of five networks each outputting a mean vector and diagonal covariance matrix for a Gaussian distribution. The safety critic is trained using standard deep Q-learning [71], while the shield is trained us-

ing DAgger-style imitation learning [83] with the Lagrangian penalty from Equation 6.2.

Note that, to allow for a fair comparison, we do not provide MSSC with an initial shield or environment model. Instead, learning proceeds without a shield for the first few episodes while an initial dataset is gathered. This dataset is then used to learn an environment model and MSSC proceeds as described in Algorithm 6.1. As a result, the first few episodes of training are not expected to be safe, but any safety violations incurred in those episodes *are* included in the safety results for the sake of fairness.

6.3.2 Scalability

In order to test the scalability of MSSC we attempt to use SPICE on a relatively high-dimensional benchmark taken from the Safety Gym suite [82]. However, SPICE requires a symbolic model class for which we can compute weakest preconditions. In this more complex benchmark, we find that our symbolic model class is not able to accurately represent the environment dynamics. As a result, SPICE is not able to compute safe actions from *any* starting state. By contrast, MSSC uses neural networks both to model environment dynamics and choose safe backup actions. Because of this increased expressivity, MSSC is able to learn a precise environment model along with an accurate safety critic and backup policy.

Benchmark	CPO	CSC-MBPO	SPICE	MSSC
acc	176	14	18	33
noisy-road-2d	172	19	15	6
pendulum	1077	136	64	29
road-2d	78	25	24	0

Table 6.1: Total safety violations.

6.3.3 Safety

Somewhat surprisingly, in addition to scaling more effectively than SPICE, MSSC is also better able to maintain safety on three of the four benchmarks. Table 6.1 shows the number of safety violations during training for the different techniques. Note that MSSC is actually at a disadvantage compared to CSC-MBPO and SPICE because both of those techniques use primarily simulated data for training, while MSSC uses simulated data only to generate shields, and uses real data for all policy optimization. Even so, MSSC is safer than SPICE and CSC-MBPO on the majority of benchmarks and still much safer than CPO on the last one.

One other interesting result not captured in Table 6.1 is the distribution of unsafe behavior in the different benchmarks. Nearly all of the safety violations incurred by MSSC happen before the first model construction phase. Recall that in this early part of the training process, MSSC cannot provide safety guarantees. After model construction and shield training, only acc exhibits any safety violations at all, with a total of four. This suggests that MSSC would benefit more than other approaches from a predefined environment model or initial safe policy which could enforce safety in the first few

episodes.

Intuitively, there are a few reasons MSSC might be safer than SPICE. First, MSSC uses a neural model of safety which considers the long-term impact of potential action. By contrast, SPICE uses a bounded-time weakest precondition approach. As a result, SPICE may end up in situations from which there is no sequence of safe actions. Second, MSSC models the environment using an ensemble of neural networks, where as SPICE uses a simplified model class to allow for formal analysis. This allows MSSC to more precisely model future states which may yield a better safety analysis. Notice that these two explanations are similar to the comparison between SPICE and CSC from Section 5.5.4. This suggests that MSSC is combining the best aspects of SPICE with the best aspects of CSC.

6.3.4 Performance

MSSC is able to achieve similar reward to prior approaches on most benchmarks. Table 6.2 shows the average final reward achieved by different tools. Notiec that, except for noisy-road-2d, MSSC achieves higher reward than SPICE and CSC-MBPO, although not as high as CPO. This is likely because MSSC uses more permissive neural models of safety and does not rely on models for policy optimization.

Benchmark	CPO	CSC-MBPO	SPICE	MSSC
acc	536	470	445	564
noisy-road-2d	-55	-330	-326	-561
pendulum	-0.005	-0.321	-0.371	-0.043
road-2d	-50	-521	-527	-510

Table 6.2: Average final rewards.

Chapter 7

Related Work

7.1 Robustness and Verification of Neural Networks

Adversarial Examples and Robustness. Szegedy et al. [95] first showed that neural networks are vulnerable to small perturbations on inputs. It has since been shown that such examples can be exploited to attack machine learning systems in safety-critical applications such as autonomous robotics [70] and malware classification [41].

Bastani et al. [12] formalized the notion of local robustness in neural networks and defined metrics to evaluate the robustness of a neural network. Subsequent work has introduced other notions of robustness [40, 51].

Many recent papers have studied the construction of adversarial counterexamples [39,42,57,66,68,74,85,96]. These approaches are based on various forms of gradient-based optimization, for example L-BFGS [95], FGSM [39] and PGD [68]. While the tool described in Chapter 2 uses the PGD method, we could in principle also use (and benefit from advances in) alternative gradient-based optimization methods.

Verification of Neural Networks. Scheibler et al. [87] used bounded model checking to verify safety of neural networks. Katz et al. [51] developed the Reluplex decision procedure extending the Simplex algorithm to verify robustness and safety properties of feedforward networks with ReLU units. Huang et al. [47] showed a verification framework, based on an SMT solver, which verified robustness with respect to a certain set of functions that can manipulate the input. A few recent papers [25, 65, 98] use Mixed Integer Linear Programming (MILP) solvers to verify local robustness properties of neural networks. These methods do not use abstraction and do not scale very well, but combining these techniques with abstraction is an interesting area of future work.

The earliest effort on neural network verification to use abstraction was by Pulina and Tacchella [80] — in fact, similar to Chapter 2, they considered an abstraction-refinement approach to solve this problem. However, their approach represents abstractions using general linear arithmetic formulas and uses a decision procedure to perform verification and counterexample search. Their approach was shown to be successful for a network with only 6 neurons, so it does not have good scalability properties. More recently, Gehr et al. [35] presented the AI^2 system for abstract interpretation of neural networks. Unlike our work, AI^2 is incomplete and cannot produce concrete counterexamples. The most closely related approach from prior work is ReluVal [100], which performs abstract interpretation using symbolic intervals. The two key differences between ReluVal and our work are that Charon couples abstract

interpretation with optimization-based counterexample search and learns verification policies from data. As demonstrated in Section 2.6, both of these ideas have a significant impact on our empirical results.

Learning to Verify. The use of data-driven learning in neural network verification is, so far as we know, new. However, there are many papers [34,46,62, 88, 89] on the use of such learning in traditional software verification. While most of these efforts learn proofs from execution data for specific programs, there are a few efforts that seek to learn optimal instantiations of parameterized abstract domains from a corpus of training problems [62, 75]. The most relevant work in this space is by Oh et al. [75], who use Bayesian optimization to adapt a parameterized abstract domain. The abstract domain in that work is finite, and the Bayesian optimizer is only used to adjust the context-sensitivity and flow-sensitivity of the analysis. In contrast, Chapter 2 handles real-valued data and a possibly infinite space of strategies.

7.2 Shielding and Verified RL

There is a growing literature on safety in RL [33]. Approaches here can be classified on basis of whether safety is guaranteed during learning or deployment. Revel, and, for example, CPO [3], were designed to enforce safety during training. Another way to categorize approaches is by whether their guarantees are probabilistic (or in expectation) or worst-case. Most approaches [3, 20, 73] are in the former category; however, Revel and prior

approaches based on verified monitors [5, 30, 31] are in the latter camp. Now we discuss in more detail three especially related threads of work.

Safety via Shielding These approaches rely on a definition of error states or fatal transitions to guarantee safety and have been used extensively in both RL and control theory [4,5,20,30,31,37,79,109]. The approach from Chapters 4 and 5 follows this general framework, but crucially introduces a mechanism to improve the shielded-policy during training. This is achieved by projecting the neural policy onto the shielded policy space. The idea of synthesizing a shield to imitate a neural policy has been explored in recent work [13, 109]. However, these approaches only generated the shield after training, so there are no guarantees about safety during training.

Formal Verification of Reinforcement Learning There is a growing literature on the verification of worst-case properties of neural networks [7, 35, 51, 52, 100]. In particular, a few recent papers [48, 91] target the verification of neural policies for autonomous agents. However, performing such verification inside a learning loop is computationally infeasible — in fact, state-of-the-art techniques failed to verify a single network from the benchmarks described in Chapter 4 within half an hour. An alternative class of approaches uses either a *nominal* environment model [28, 53] or a user-provided safe policy as a starting point for safe learning [19, 20]. In both cases, these techniques require a predefined model of the dynamics of the environment. In contrast,

Chapter 5 does not require the user to specify any model of the environment, so it can be applied to a much broader set of problems.

7.3 Statistical Safe Learning

A variety of recent work has focused on applying statistical approaches to safe exploration and safe learning [3, 14, 15, 23, 58, 60, 64]. These approaches maintain an environment model and then use a variety of different statistical approaches to attempt to generate safe policies. Some work with cost constraints, in which there is a continuous cost function which must be kept below a certain threshold, and others are based on boolean safety signals. In both cases, these techniques suffer from two sources of unsafe behavior: (1) the model they maintain can be inaccurate and (2) the statistical approaches they use to generate policies may fail. Compared to these approaches, shielding eliminates the second source of error, leading to much safer behavior in practice. Moreover, in the presence of a known environment model (as in Chapter 4), shielding can fully eliminate both sources of error in order to generate provably safe policies.

Many approaches to the safe reinforcement learning problem provide statistical bounds on the system safety [3,64,67,86,103,108]. These approaches maintain an environment model and then use a variety of statistical techniques to generate a policy which is likely to be safe with respect to the environment model. This leads to two potential sources of unsafe behavior: the policy may be unsafe with respect to the model, or the model may be an inaccurate rep-

resentation of the environment. Compared to these approaches, we eliminate the first source of error by always generating policies that are *guaranteed* to be safe with respect to an environment model. We show in our experiments that this drastically reduces the number of safety violations encountered in practice. Some techniques use a learned model together with a linearized cost model to provide bounds, similar to Chapter 5 [23, 61]. However, in these works, the system only looks ahead one time step and relies on the assumption that the cost signal cannot have too much inertia. Chapter 5 alleviates this problem by providing a way to look ahead several time steps to achieve a more precise safety analysis.

A subset of the statistical approaches are tools that maintain neural models of the cost function in order to intervene in unsafe behavior [15, 102, 104]. These approaches maintain a critic network which represents the long-term cost of taking a particular action in a particular state. However, because of the amount of data needed to train neural networks accurately, these approaches suffer from a need to collect data in several unsafe trajectories in order to build the cost model. The symbolic approach of Chapter 5 is more data-efficient, allowing the system to avoid safety violations more often in practice. This is borne out by the experiments in Section 5.5.

Chapter 8

Conclusion

As machine learning has shown its power in a wide variety of applications, it has also shown its fragility and opacity. New techniques and new algorithms are needed to ensure that as we deploy neural networks in real-world systems, we can understand and constrain how these networks will behave. This dissertation shows how ideas from classical program analysis can be extended and adapted to handle systems with machine learning components. Moreover, we tightly weaves traditional program analysis together with machine learning, allowing the two fields to support each other. This leads to systems which achieve the performance of machine learning while also maintaining the safety of traditional programs.

In the context of robustness, this takes the form of a program analysis framework which is both supported by and applied to machine learning systems. Specifically, we use ideas from program analysis to determine whether neural networks satisfy certain properties. At the same time, we use more traditional "shallow" machine learning approaches to develop heuristics for that analysis engine. In this work, the machine learning and program anlysis support each other, and we show experimentally that this support results in

a better tool than existing work that focused on using machine learning or program anlysis in isolation.

In the context of reinforcement learning, this dissertation focuses on training neural networks and programmatic controllers together, allowing them to work together as a single policy. We have developed (or are currently developing) several variations on this idea, focusing on different forms of programmatic controllers and different modes of interaction between the neural and programmatic components. We have looked at straightforward versions of this, where the neural and symbolic policies are completely separate, as well as more tightly integrated systems where the neural and programmatic policies are closely related. In this context, we have seen how combining traditional program synthesis with neural network training leads to controllers which achieve both high performance and safety.

The combination of programming languages with machine learning offers an approach to learning which combines the safety, sample efficiency, and interpretability of programs with the power of neural networks. In this work, we have demonstrated this synergy in a few contexts — namely robustness analysis and reinforcement learning. In both cases, machine learning approaches alone do not achieve the desirable safety properties of program languages approaches. At the same time, programming languages techniques simply do not scale to the level which is required for machine learning systems. However, in the combination of both these fields, we find the power we need to analyze or constrain machine-learning-scale systems, along with the safety

and analyzability we have come to expect from traditional programs.

Bibliography

[1] Elina: Eth library for numerical analysis.

[2] Google cloud platform (gcp). `https://cloud.google.com/`. Accessed: 2018-11-14.

[3] Joshua Achiam, David Held, Aviv Tamar, and Pieter Abbeel. Constrained policy optimization. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ICML'17, page 2231. JMLR.org, 2017.

[4] Anayo K. Akametalu, Shahab Kaynama, Jaime F. Fisac, Melanie Nicole Zeilinger, Jeremy H. Gillula, and Claire J. Tomlin. Reachability-based safe learning with gaussian processes. In *53rd IEEE Conference on Decision and Control, CDC 2014, Los Angeles, CA, USA, December 15-17, 2014*, pages 1424–1431, 2014.

[5] Mohammed Alshiekh, Roderick Bloem, Rüdiger Ehlers, Bettina Könighofer, Scott Niekum, and Ufuk Topcu. Safe reinforcement learning via shielding. In Sheila A. McIlraith and Kilian Q. Weinberger, editors, *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in*

Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018, pages 2669–2678. AAAI Press, 2018.

[6] Greg Anderson, Swarat Chaudhuri, and Isil Dillig. Guiding safe exploration with weakest preconditions. In *Proceedings of the 11th International Conference on Learning Representations*, 2023.

[7] Greg Anderson, Shankara Pailoor, Isil Dillig, and Swarat Chaudhuri. Optimization and abstraction: A synergistic approach for analyzing neural network robustness. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 731744, New York, NY, USA, 2019. Association for Computing Machinery.

[8] Greg Anderson, Abhinav Verma, Isil Dillig, and Swarat Chaudhuri. Neurosymbolic reinforcement learning with formally verified exploration. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NeurIPS'20, Red Hook, NY, USA, 2020. Curran Associates Inc.

[9] Martin Anderson, Joachim Dahl, and Lieven Vandenberghe. CVXOPT. `https://github.com/cvxopt/cvxopt`, 2022.

[10] ApolloAuto. apollo, 2017.

[11] Edoardo Bacci, Mirco Giacobbe, and David Parker. Verifying reinforcement learning up to infinity. In Zhi-Hua Zhou, editor, *Proceedings of*

the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI-21, pages 2154–2160. International Joint Conferences on Artificial Intelligence Organization, 8 2021. Main Track.

[12] Osbert Bastani, Yani Ioannou, Leonidas Lampropoulos, Dimitrios Vytiniotis, Aditya V. Nori, and Antonio Criminisi. Measuring neural net robustness with constraints. In *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*, pages 2613–2621, 2016.

[13] 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.

[14] 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.

[15] Homanga Bharadhwaj, Aviral Kumar, Nicholas Rhinehart, Sergey Levine, Florian Shkurti, and Animesh Garg. Conservative safety critics for exploration. In *International Conference on Learning Representations*, 2021.

[16] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard

Firner, Beat Flepp, Prasoon Goyal, Lawrence D. Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, Xin Zhang, Jake Zhao, and Karol Zieba. End to end learning for self-driving cars. *CoRR*, abs/1604.07316, 2016.

[17] Eric Brochu, Vlad M. Cora, and Nando De Freitas. A tutorial on bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning. abs/1012.2599, 12 2010.

[18] E. M. Bronshteyn and L. D. Ivanov. The approximation of of convex sets by polyhedra. *Sib Math J*, 16(5):852–853, September-October 1975.

[19] Richard Cheng, Gábor Orosz, Richard M. Murray, and Joel W. Burdick. End-to-end safe reinforcement learning through barrier functions for safety-critical continuous control tasks. In *Proceedings of the Thirty-Third AAAI Conference on Artificial Intelligence*, AAAI'19. AAAI Press, 2019.

[20] Yinlam Chow, Ofir Nachum, Edgar Duenez-Guzman, and Mohammad Ghavamzadeh. A lyapunov-based approach to safe reinforcement learning. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NeurIPS'18, page 81038112, Red Hook, NY, USA, 2018. Curran Associates Inc.

[21] Kurtland Chua, Roberto Calandra, Rowan McAllister, and Sergey Levine. Deep reinforcement learning in a handful of trials using probabilistic dynamics models. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman,

N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018.

[22] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Annual ACM Symposium on Principles of Programming Languages*, 1977.

[23] Gal Dalal, Krishnamurthy Dvijotham, Matej Vecerik, Todd Hester, Cosmin Paduraru, and Yuval Tassa. Safe Exploration in Continuous Action Spaces. January 2018.

[24] Edsger Wybe Dijkstra. *A discipline of programming*, volume 613924118. prentice-hall Englewood Cliffs, 1976.

[25] Souradeep Dutta, Susmit Jha, Sriram Sankaranarayanan, and Ashish Tiwari. Output range analysis for deep neural networks. *CoRR*, abs/1709.09130, 2017.

[26] Andre Esteva, Brett Kuprel, Roberto A. Novoa, Justin Ko, Susan M. Swetter, Helen M. Blau, and Sebastian Thrun. Dermatologist-level classification of skin cancer with deep neural networks. *Nature*, 542:115–118, 01 2017.

[27] Ivan Evtimov, Kevin Eykholt, Earlence Fernandes, Tadayoshi Kohno, Bo Li, Atul Prakash, Amir Rahmati, and Dawn Song. Robust physical-

world attacks on machine learning models. *CoRR*, abs/1707.08945, 2017.

[28] Jaime F. Fisac, Anayo K. Akametalu, Melanie N. Zeilinger, Shahab Kaynama, Jeremy Gillula, and Claire J. Tomlin. A general safety framework for learning-based control in uncertain robotic systems. *IEEE Transactions on Automatic Control*, 64(7):2737–2752, 2019.

[29] Message P Forum. Mpi: A message-passing interface standard. Technical report, Knoxville, TN, USA, 1994.

[30] Nathan Fulton and André Platzer. Safe reinforcement learning via formal methods: Toward safe control through proof and learning. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[31] Nathan Fulton and André Platzer. Verifiably safe off-model reinforcement learning. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 413–430. Springer, 2019.

[32] Sicun Gao, Jeremy Avigad, and Edmund M. Clarke. δ-complete decision procedures for satisfiability over the reals. In *Proceedings of the 6th International Joint Conference on Automated Reasoning*, IJCAR'12, pages 286–300, Berlin, Heidelberg, 2012. Springer-Verlag.

[33] Javier Garcıa and Fernando Fernández. A comprehensive survey on safe reinforcement learning. *Journal of Machine Learning Research*,

16(1):1437–1480, 2015.

[34] Pranav Garg, Christof Löding, P Madhusudan, and Daniel Neider. Ice: A robust framework for learning invariants. In *International Conference on Computer Aided Verification*, pages 69–87. Springer, 2014.

[35] Timon Gehr, Matthew Mirman, Dana Drachsler-Cohen, Petar Tsankov, Swarat Chaudhuri, and Martin Vechev. Ai2: Safety and robustness certification of neural networks with abstract interpretation. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 3–18. IEEE, 2018.

[36] Khalil Ghorbal, Eric Goubault, and Sylvie Putot. The zonotope abstract domain taylor1+. In *Proceedings of the 21st International Conference on Computer Aided Verification*, CAV '09, pages 627–633, Berlin, Heidelberg, 2009. Springer-Verlag.

[37] Jeremy H. Gillula and Claire J. Tomlin. Guaranteed safe online learning via reachability: tracking a ground target using a quadrotor. In *IEEE International Conference on Robotics and Automation, ICRA 2012, 14-18 May, 2012, St. Paul, Minnesota, USA*, pages 2723–2730, 2012.

[38] Yuan Gong and Christian Poellabauer. An overview of vulnerabilities of voice controlled systems. *arXiv preprint arXiv:1803.09156*, 2018.

[39] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. In *International Conference on*

Learning Representations, 2015.

[40] Divya Gopinath, Guy Katz, Corina S Pasareanu, and Clark Barrett. Deepsafe: A data-driven approach for checking adversarial robustness in neural networks. *arXiv preprint arXiv:1710.00486*, 2017.

[41] Kathrin Grosse, Nicolas Papernot, Praveen Manoharan, Michael Backes, and Patrick McDaniel. Adversarial examples for malware detection. In *European Symposium on Research in Computer Security*, pages 62–79. Springer, 2017.

[42] Kathrin Grosse, Nicolas Papernot, Praveen Manoharan, Michael Backes, and Patrick D. McDaniel. Adversarial perturbations against deep neural networks for malware classification. *CoRR*, abs/1606.04435, 2016.

[43] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 1861–1870. PMLR, 10–15 Jul 2018.

[44] Tuomas Haarnoja, Aurick Zhou, Kristian Hartikainen, George Tucker, Sehoon Ha, Jie Tan, Vikash Kumar, Henry Zhu, Abhishek Gupta, Pieter Abbeel, and Sergey Levine. Soft actor-critic algorithms and applications, 2018.

[45] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, June 2016.

[46] Kihong Heo, Hakjoo Oh, and Hongseok Yang. Learning a variable-clustering strategy for octagon from labeled data generated by a static analysis. In *International Static Analysis Symposium*, pages 237–256. Springer, 2016.

[47] Xiaowei Huang, Marta Kwiatkowska, Sen Wang, and Min Wu. Safety verification of deep neural networks. *CoRR*, abs/1610.06940, 2016.

[48] Radoslav Ivanov, James Weimer, Rajeev Alur, George J Pappas, and Insup Lee. Verisig: verifying safety properties of hybrid systems with neural network controllers. In *Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control*, pages 169–178, 2019.

[49] Michael Janner, Justin Fu, Marvin Zhang, and Sergey Levine. *When to Trust Your Model: Model-Based Policy Optimization*. Curran Associates Inc., Red Hook, NY, USA, 2019.

[50] Kyle D. Julian, Jessica Lopez, Jeffrey S. Brush, Michael P. Owen, and Mykel J. Kochenderfer. Policy compression for aircraft collision avoidance systems. In *2016 IEEE/AIAA 35th Digital Avionics Systems Conference (DASC)*, pages 1–10, Sept 2016.

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

[52] Guy Katz, Derek A. Huang, Duligur Ibeling, Kyle Julian, Christopher Lazarus, Rachel Lim, Parth Shah, Shantanu Thakoor, Haoze Wu, Aleksandar Zeljic, David L. Dill, Mykel J. Kochenderfer, and Clark W. Barrett. The marabou framework for verification and analysis of deep neural networks. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I*, volume 11561 of *Lecture Notes in Computer Science*, pages 443–452. Springer, 2019.

[53] Torsten Koller, Felix Berkenkamp, Matteo Turchetta, and Andreas Krause. Learning-based model predictive control for safe exploration. In *2018 IEEE Conference on Decision and Control (CDC)*, pages 6059–6066, 2018.

[54] Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical report, 2009.

[55] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances*

in Neural Information Processing Systems 25, pages 1097–1105. Curran Associates, Inc., 2012.

[56] Aviral Kumar, Aurick Zhou, George Tucker, and Sergey Levine. Conservative q-learning for offline reinforcement learning. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 1179–1191. Curran Associates, Inc., 2020.

[57] Alexey Kurakin, Ian J. Goodfellow, and Samy Bengio. Adversarial machine learning at scale. *CoRR*, abs/1611.01236, 2016.

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

[59] Yann Lecun, Lon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*, pages 2278–2324, 1998.

[60] Xiao Li and Calin Belta. Temporal Logic Guided Safe Reinforcement Learning Using Control Barrier Functions. *arXiv:1903.09885 [cs, stat]*, March 2019. arXiv: 1903.09885.

[61] Yutong Li, Nan Li, H. Eric Tseng, Anouck Girard, Dimitar Filev, and Ilya Kolmanovsky. Safe reinforcement learning using robust action governor. In Ali Jadbabaie, John Lygeros, George J. Pappas, Pablo A. Par-

rilo, Benjamin Recht, Claire J. Tomlin, and Melanie N. Zeilinger, editors, *Proceedings of the 3rd Conference on Learning for Dynamics and Control*, volume 144 of *Proceedings of Machine Learning Research*, pages 1093–1104. PMLR, 07 – 08 June 2021.

[62] Percy Liang, Omer Tripp, and Mayur Naik. Learning minimal abstractions. In *POPL*, volume 46, pages 31–42. ACM, 2011.

[63] 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.

[64] Zuxin Liu, Hongyi Zhou, Baiming Chen, Sicheng Zhong, Martial Hebert, and Ding Zhao. Constrained model-based reinforcement learning with robust cross-entropy method, 2020.

[65] Alessio Lomuscio and Lalit Maganti. An approach to reachability analysis for feed-forward relu neural networks. *CoRR*, abs/1706.07351, 2017.

[66] Chunchuan Lyu, Kaizhu Huang, and Hai-Ning Liang. A unified gradient regularization family for adversarial examples. In *2015 IEEE International Conference on Data Mining, ICDM 2015, Atlantic City, NJ, USA, November 14-17, 2015*, pages 301–309, 2015.

[67] Yecheng Jason Ma, Andrew Shen, Osbert Bastani, and Dinesh Jayaraman. Conservative and adaptive penalty for model-based safe reinforce-

ment learning, 2021.

[68] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards deep learning models resistant to adversarial attacks. In *International Conference on Learning Representations (ICLR)*, 2018.

[69] Ruben Martinez-Cantin. Bayesopt: A bayesian optimization library for nonlinear optimization, experimental design and bandits. *Journal of Machine Learning Research*, 15:3915–3919, 2014.

[70] Marco Melis, Ambra Demontis, Battista Biggio, Gavin Brown, Giorgio Fumera, and Fabio Roli. Is deep learning safe for robot vision? adversarial examples against the icub humanoid. In *Computer Vision Workshop (ICCVW), 2017 IEEE International Conference on*, pages 751–759. IEEE, 2017.

[71] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. 2013. cite arxiv:1312.5602Comment: NIPS Deep Learning Workshop 2013.

[72] Jonas Mockus. *Bayesian Heuristic Approach to Discrete and Global Optimization: Algorithms, Visualization, Software, and Applications*. Springer-Verlag, Berlin, Heidelberg, 2010.

[73] Teodor Mihai Moldovan and Pieter Abbeel. Safe exploration in markov decision processes. In *Proceedings of the 29th International Conference on Machine Learning, ICML 2012, Edinburgh, Scotland, UK, June 26 - July 1, 2012*, 2012.

[74] Anh Mai Nguyen, Jason Yosinski, and Jeff Clune. Deep neural networks are easily fooled: High confidence predictions for unrecognizable images. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2015, Boston, MA, USA, June 7-12, 2015*, pages 427–436, 2015.

[75] Hakjoo Oh, Hongseok Yang, and Kwangkeun Yi. Learning a strategy for adapting a program analysis via bayesian optimisation. In *ACM SIGPLAN Notices*, volume 50, pages 572–588. ACM, 2015.

[76] Nicolas Papernot, Patrick D. McDaniel, and Ian J. Goodfellow. Transferability in machine learning: from phenomena to black-box attacks using adversarial samples. *CoRR*, abs/1605.07277, 2016.

[77] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. Deepxplore: Automated whitebox testing of deep learning systems. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 1–18, New York, NY, USA, 2017. ACM.

[78] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. Towards practical verification of machine learning: The case of computer vision systems. *CoRR*, abs/1712.01785, 2017.

[79] Theodore J. Perkins and Andrew G. Barto. Lyapunov design for safe reinforcement learning. *J. Mach. Learn. Res.*, 3:803–832, 2002.

[80] Luca Pulina and Armando Tacchella. An abstraction-refinement approach to verification of artificial neural networks. In *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*, pages 243–257, 2010.

[81] Carl Edward Rasmussen and Christopher K. I. Williams. *Gaussian Processes for Machine Learning*. The MIT Press, 2006.

[82] Alex Ray, Joshua Achiam, and Dario Amodei. Benchmarking Safe Exploration in Deep Reinforcement Learning. page 25.

[83] Stephane Ross, Geoffrey Gordon, and Drew Bagnell. A reduction of imitation learning and structured prediction to no-regret online learning. In Geoffrey Gordon, David Dunson, and Miroslav Dudk, editors, *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, volume 15 of *Proceedings of Machine Learning Research*, pages 627–635, Fort Lauderdale, FL, USA, 11–13 Apr 2011. PMLR.

[84] Jason Rudy. PyEarth. `https://github.com/scikit-learn-contrib/py-earth`, 2013.

[85] Sara Sabour, Yanshuai Cao, Fartash Faghri, and David J. Fleet. Adversarial manipulation of deep representations. *CoRR*, abs/1511.05122,

2015.

[86] Harsh Satija, Philip Amortila, and Joelle Pineau. Constrained markov decision processes via backward value functions. In *Proceedings of the 37th International Conference on Machine Learning*, ICML'20. JMLR.org, 2020.

[87] Karsten Scheibler, Leonore Winterer, Ralf Wimmer, and Bernd Becker. Towards verification of artificial neural networks. In *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen, MBMV 2015, Chemnitz, Germany, March 3-4, 2015.*, pages 30–40, 2015.

[88] Rahul Sharma, Saurabh Gupta, Bharath Hariharan, Alex Aiken, and Aditya V Nori. Verification as learning geometric concepts. In *International Static Analysis Symposium*, pages 388–411. Springer, 2013.

[89] Xujie Si, Hanjun Dai, Mukund Raghothaman, Mayur Naik, and Le Song. Learning loop invariants for program verification. In *Proceedings of the Thirty-second Conference on Neural Information Processing Systems*, 2018.

[90] Gagandeep Singh, Markus Püschel, and Martin Vechev. Fast polyhedra abstract domain. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 46–59, New York, NY, USA, 2017. ACM.

[91] Xiaowu Sun, Haitham Khedr, and Yasser Shoukry. Formal verification of neural network controlled autonomous systems. In *Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control*, pages 147–156, 2019.

[92] Richard S. Sutton. Integrated architecture for learning, planning, and reacting based on approximating dynamic programming. In *Proceedings of the Seventh International Conference (1990) on Machine Learning*, page 216224, San Francisco, CA, USA, 1990. Morgan Kaufmann Publishers Inc.

[93] Richard S. Sutton. Planning by incremental dynamic programming. In *Proceedings of the Eighth International Conference on Machine Learning*, ML'91, page 353357, San Francisco, CA, USA, 1991. Morgan Kaufmann Publishers Inc.

[94] Richard S. Sutton, David McAllester, Satinder Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Proceedings of the 12th International Conference on Neural Information Processing Systems*, NIPS'99, page 10571063, Cambridge, MA, USA, 1999. MIT Press.

[95] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian J. Goodfellow, and Rob Fergus. Intriguing properties of neural networks. *CoRR*, abs/1312.6199, 2013.

[96] Pedro Tabacof and Eduardo Valle. Exploring the space of adversarial images. In *2016 International Joint Conference on Neural Networks, IJCNN 2016, Vancouver, BC, Canada, July 24-29, 2016*, pages 426–433, 2016.

[97] Pranjal Tandon. PyTorch Soft Actor-Critic. `https://github.com/pranz24/pytorch-soft-actor-critic`, 2018.

[98] Vincent Tjeng and Russ Tedrake. Verifying neural networks with mixed integer programming. *CoRR*, abs/1711.07356, 2017.

[99] Abhinav Verma, Hoang Le, Yisong Yue, and Swarat Chaudhuri. Imitation-projected programmatic reinforcement learning. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

[100] Shiqi Wang, Kexin Pei, Justin Whitehouse, Junfeng Yang, and Suman Jana. Formal security analysis of neural networks using symbolic intervals. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 1599–1614, 2018.

[101] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Lukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto

Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean. Google's neural machine translation system: Bridging the gap between human and machine translation. *CoRR*, abs/1609.08144, 2016.

[102] Qisong Yang, Thiago D. Simo, Simon H Tindemans, and Matthijs T. J. Spaan. Wcsac: Worst-case soft actor critic for safety-constrained reinforcement learning. *Proceedings of the AAAI Conference on Artificial Intelligence*, 35(12):10639–10646, May 2021.

[103] Tsung-Yen Yang, Justinian Rosca, Karthik Narasimhan, and Peter J Ramadge. Projection-based constrained policy optimization. *arXiv preprint arXiv:2010.03152*, 2020.

[104] Dongjie Yu, Haitong Ma, Shengbo Li, and Jianyu Chen. Reachability constrained reinforcement learning. In Kamalika Chaudhuri, Stefanie Jegelka, Le Song, Csaba Szepesvari, Gang Niu, and Sivan Sabato, editors, *Proceedings of the 39th International Conference on Machine Learning*, volume 162 of *Proceedings of Machine Learning Research*, pages 25636–25655. PMLR, 17–23 Jul 2022.

[105] Xiaoyong Yuan, Pan He, Qile Zhu, Rajendra Rana Bhat, and Xiaolin Li. Adversarial examples: Attacks and defenses for deep learning. *CoRR*, abs/1712.07107, 2017.

[106] Zhenlong Yuan, Yongqiang Lu, Zhaoguo Wang, and Yibo Xue. Droid-sec: Deep learning in android malware detection. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 371–372. ACM, 2014.

[107] Zhenlong Yuan, Yongqiang Lu, and Yibo Xue. Droiddetector: android malware characterization and detection using deep learning. *Tsinghua Science and Technology*, 21(1):114–123, Feb 2016.

[108] Yiming Zhang, Quan Vuong, and Keith Ross. First order constrained optimization in policy space. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 15338–15349. Curran Associates, Inc., 2020.

[109] 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.