Optimization + Abstraction: A Synergistic Approach for Analyzing Neural Network Robustness

Greg Anderson
The University of Texas at Austin
Austin, Texas, USA
ganderso@cs.utexas.edu

Shankara Pailoor
The University of Texas at Austin
Austin, Texas, USA
spailoor@cs.utexas.edu

Isil Dillig
The University of Texas at Austin
Austin, Texas, USA
isil@cs.utexas.edu

Swarat Chaudhuri
Rice University
Houston, Texas, USA
swarat@rice.edu

Abstract

In recent years, the notion of *local robustness* (or *robustness* for short) has emerged as a desirable property of deep neural networks. Intuitively, robustness means that small perturbations to an input do not cause the network to perform misclassifications. In this paper, we present a novel algorithm for verifying robustness properties of neural networks. Our method synergistically combines gradient-based optimization methods for counterexample search with abstraction-based proof search to obtain a sound and (δ-)complete decision procedure. Our method also employs a data-driven approach to learn a verification policy that guides abstract interpretation during proof search. We have implemented the proposed approach in a tool called Charon and experimentally evaluated it on hundreds of benchmarks. Our experiments show that the proposed approach significantly outperforms three state-of-the-art tools, namely AI^2, Reluplex, and Reluval.

CCS Concepts • **Theory of computation → Abstraction**; • **Computing methodologies** → *Neural networks*;

Keywords Machine learning, Abstract Interpretation, Optimization, Robustness

ACM Reference Format:
Greg Anderson, Shankara Pailoor, Isil Dillig, and Swarat Chaudhuri. 2019. Optimization + Abstraction:, A Synergistic Approach for Analyzing Neural Network Robustness. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '19), June 22–26, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3314221.3314614

PLDI '19, June 22–26, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6712-7/19/06…$15.00
https://doi.org/10.1145/3314221.3314614

1 Introduction

In recent years, deep neural networks (DNNs) have gained enormous popularity for a wide spectrum of applications, ranging from image recognition[21, 28] and malware detection [57, 58] to machine translation[55]. Due to their surprising effectiveness in practice, deep learning has also found numerous applications in safety-critical systems, including self-driving cars [3, 5], unmanned aerial systems [24], and medical diagnosis [9].

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 [10, 40]. For instance, Figure 1 shows two adversarial examples in the context of speech recognition and image classification. As shown in the top half of Figure 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 [16]. 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 [56]. Prior work [4, 14, 25, 41, 42] 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 x as having label y. Then, local robustness requires that all inputs x' that are "very similar" [1] to x are also classified as having the same label y.

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

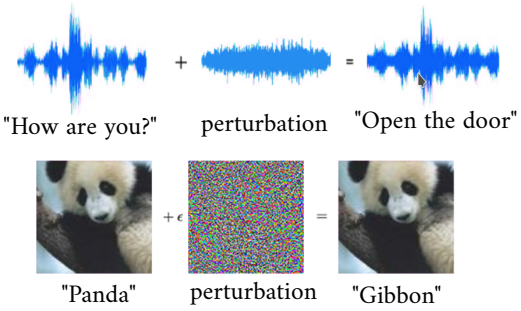

Figure 1. Small perturbations of the input cause the sound wave and the image to be misclassified.

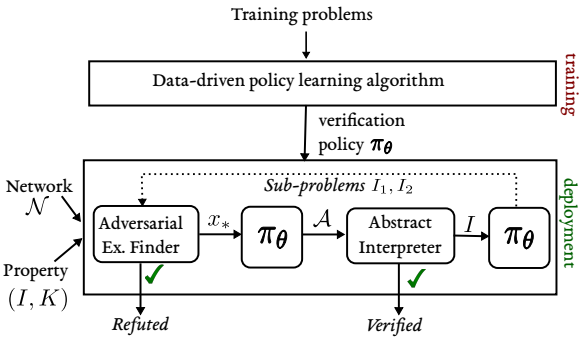

Figure 2. Schematic overview of our approach.

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) [34] and Fast Gradient Sign Method (FGSM) [17]. A second category aims to *prove* network robustness using symbolic methods ranging from SMT-solving [25] to abstract interpretation [14, 54]. 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 [14] (i.e., suffer from false positives) or do not scale well [25].

In this paper, 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 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 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 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 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 [12]. 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 x_*, this means that 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 [30] and CIFAR [27] 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 paper 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.
- 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.

Organization. The rest of this paper is organized as follows. In Section 2, we provide necessary background on neural networks, robustness, and abstract interpretation of neural networks. In Section 3, we present our algorithm for checking robustness, given a verification policy (i.e., the deployment phase). Section 4 describes our data-driven approach for learning a useful verification policy from training data (i.e., the training phase), and Section 5 discusses the theoretical properties of our algorithm. Finally, Sections 6 and 7 describe our implementation and experimental evaluation, Section 8 discusses related work, and Section 9 concludes.

2 Background

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

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

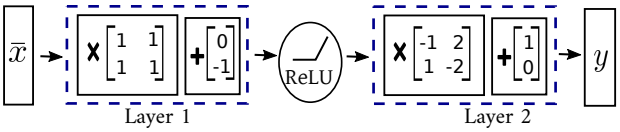

Figure 3. A feedforward network implementing XOR

2.1 Neural Networks

A neural network is 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 [3].

For the purposes of this work, we think of each layer L_i as an affine transformation $(W, \overline{b})$ where W is a *weight matrix* and $\overline{b}$ is a *bias vector*. Thus, the output of the i'th layer is computed as $\overline{y} = W\overline{x} + \overline{b}$. We note that both *fully-connected* as well as *convolutional layers* can be expressed as affine transformations [14]. 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 work, we consider networks used for classification tasks. That is, given some input $x \in \mathbb{R}^n$, we wish to put x into one of m *classes*. The output of the network $y \in \mathbb{R}^m$ is interpreted as a vector of *scores*, one for each class. Then x is put into the class with the highest score. More formally, given some input x, we say the network $\mathcal{N}$ assigns x to a class K if $(\mathcal{N}(x))_K > (\mathcal{N}(x))_j$ for all $j \neq K$.

Example 2.1. Figure 3 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.

2.2 Robustness

(Local) robustness [4] 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.

[3] Recall that a function is Lipschitz-continuous if there exists a positive real constant M such that, for all x_1, x_2, we have $|f(x_1) - f(x_2)| \leq M|x_1 - x_2|$

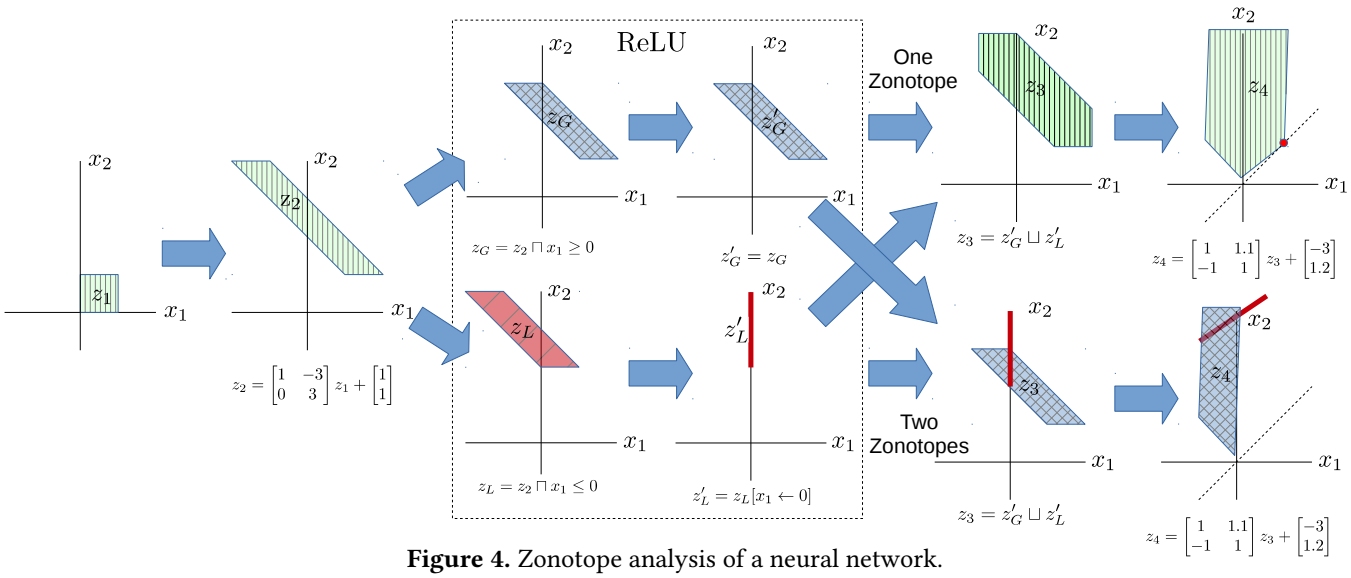

Figure 4. Zonotope analysis of a neural network.

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 $x \in I$, we have $(\mathcal{N}(x))_K > (\mathcal{N}(x))_j$ for all $j \neq K$.

Example 2.2. Consider the following network with two layers, i.e., $\mathcal{N}(x) = W_2(\text{ReLU}(W_1 x + b_1)) + b_2$ where:

$$W_1 = \begin{bmatrix} 1 \\ 2 \end{bmatrix} \quad b_1 = \begin{bmatrix} -1 \\ 1 \end{bmatrix} \quad W_2 = \begin{bmatrix} 2 & 1 \\ -1 & 1 \end{bmatrix} \quad b_2 = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

For the input $x = 0$, we have $\mathcal{N}(0) = [1\ 3]^\top$; thus, the network outputs label 1 for input 0. Let $I = [-1, 1]$ and $K = 1$. Then for all $x \in I$, the output of $\mathcal{N}$ is of the form $[a+1\ a+2]^T$ 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 $[-1, 2]$.

2.3 Abstract Interpretation for Neural Networks

In this paper, we build on the prior AI^2 work [14] for analyzing neural networks using the framework of *abstract interpretation* [7]. AI^2 allows analyzing neural networks using a variety of numeric abstract domains, including intervals (boxes) [7], polyhedra [49], and zonotopes [15]. In addition, AI^2 also supports *bounded powerset domains* [7], 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 zonotope domain with two disjuncts but not with intervals or plain zonotopes:

Example 2.3. Consider a network defined as:

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

As in the previous example, the first index in the output vector corresponds to class A and the second index corresponds to class B. Now, suppose we want to verify that for all $x \in [0, 1]^2$, the network assigns class B to 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 4. 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 $x_1 \geq 0$ and the red (diagonally striped) one for which $x_1 \leq 0$. The ReLU transforms the red piece into a line. (We omit the ReLU over 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.

3 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:

$$x_* = \arg\min_{x \in I} (\mathcal{F}(x)) \tag{1}$$

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

$$\mathcal{F}(x) = (\mathcal{N}(x))_K - \max_{j \neq K} (\mathcal{N}(x))_j \tag{2}$$

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 x, then there exists some class which has a greater (or equal) score than the target class, so point x constitutes a true adversarial counterexample.

The optimization problem from Eq. 1 is clearly useful for searching for counterexamples to the robustness property. However, even if the solution x_* is not a true counterexample (i.e., $\mathcal{F}(x_*) > 0$), 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 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. As mentioned in Section 1, 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

Algorithm 1 The main algorithm

```
1:  procedure Verify(N, I, K, π_θ)
    input: A network N, robustness property (I, K) and
    verification policy π_θ = (π^α_θ, π^I_θ)
    output: Counterexample if N is not robust, or Verified.
2:      x_* ← Minimize(I, F)
3:      if F(x_*) ≤ 0 then
4:          return x_*
5:      A ← π^α_θ(N, I, K, x_*)
6:      if Analyze(N, I, K, A) = Verified then
7:          return Verified
8:      (I_1, I_2) ← π^I_θ(N, I, K, x_*)
9:      r_1 ← Verify(N, I_1, K, π_θ)
10:     if r_1 ≠ Verified then
11:         return r_1
12:     return Verify(N, I_2, K, π_θ)
```

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 4, we explain how the parameters of these policy functions are learned from data.

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 Eq. 1 using the well-known *projected gradient descent (PGD)* technique. If $\mathcal{F}(x_*)$ is non-positive, we have found a true counterexample, so the algorithm produces 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, 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^α_θ to choose a sensible abstract domain to use when attempting to verify the property. Specifically, the domain policy π^α_θ takes as input the network $\mathcal{N}$, the robustness specification (I, K), and the solution 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 $I = I_1 \uplus I_2$. 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

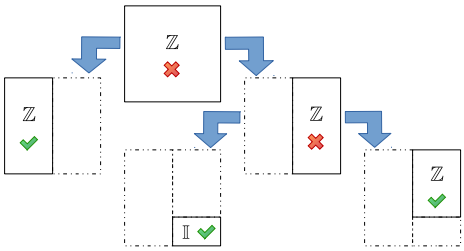

Figure 5. The splits chosen for Example 3.1.

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 utilize our *partition policy* π^I_θ to make this decision. Similar to the domain policy, π^I_θ 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.

Example 3.1. Consider the XOR network from Figure 3 and the robustness property $([0.3, 0.7]^2, 1)$. That is, for all inputs $\overline{x}$ with $0.3 \leq x_1, x_2 \leq 0.7$, $\overline{x}$ should be assigned to class 1 (assume classes are zero-indexed). We now illustrate how Algorithm 1 verifies this property using the plain interval and zonotope abstract domains. The process is illustrated in Figure 5, 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 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^α_θ 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^I_θ 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 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^α_θ 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^I_θ 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^α_θ 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.

4 Learning a Verification Policy

As described in Section 3, our decision procedure for checking robustness uses a verification policy $\pi_\theta = (\pi^\alpha_\theta, \pi^I_\theta)$ 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.

4.1 Policy Representation

In this work, we implement verification policies π^α_θ and π^I_θ using a function of the following shape:

$$\varphi(\theta\, \rho(\mathcal{N}, I, K, x_*)) \tag{3}$$

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^α_θ and the two subregions for π^I_θ), 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 $\iota = (\mathcal{N}, I, K, x_*)$ 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 x_*, (b) where x_* falls in the input space, and (c) the size of the input space. Intuitively, we expect that (a) is useful because as 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 6, 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.

4.2 Learning using Bayesian Optimization

As made evident by Eq. 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 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 $\overline{x^*} \in \mathbb{R}^n$ 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 $\overline{x}$ according to $\mathcal{A}$, evaluates F at $\overline{x}$, and updates the statistical model $\mathcal{M}$ based on the observation $F(\overline{x})$. 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 [37] 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:

$$F(\theta) = -\sum_{s \in S} cost_\theta(s)$$

where:

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

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. [4]

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* [44] as our surrogate model and use *expected improvement* [6] for the acquisition function.

5 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 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 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 1 always terminates or whether it has any completeness guarantees.

Our first observation is that the Verify procedure, *exactly* as presented in Algorithm 1, does not have termination guarantees under realistic assumptions about the optimization procedure used for finding adversarial counterexamples.

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

Specifically, if the procedure Minimize invoked at line 2 of Algorithm 1 returned a *global* minimum, then we could indeed guarantee termination. [5] However, since gradient-based optimization procedures do not have this property, Algorithm 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 1.

To guarantee termination of our verification algorithm, we will make the following slight change to line 3 of Algorithm 1: Rather than checking $\mathcal{F}(x_*) \leq 0$ (for $\mathcal{F}$ as defined in Eq. 2) we will instead check:

$$\mathcal{F}(x_*) \leq \delta \tag{4}$$

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 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 5.1. For any set $X \subseteq \mathbb{R}^n$, its *diameter* $D(X)$ is defined as $\sup\{\|x_1 - x_2\|_2 \mid x_1, x_2 \in X\}$ 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 1. *There exists some $\lambda \in (0,1)$ such that for any network $\mathcal{N}$, input region I, and point $x_* \in I$, if $\pi^I(N, I, x_*) = (I_1, I_2)$, 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. *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 5.2. *Consider the variant of Algorithm 1 where the predicate at line 3 is replaced with Eq. 4. Then, if the input region has finite diameter, the verification algorithm always terminates under Assumptions 1 and 2.* [6]

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

[6] Proofs for all theorems can be found in the appendix.

In addition to termination, our small modification to Algorithm 1 also ensures a property called δ-completeness [12]. 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 5.3. 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 5.4. *Consider the variant of Algorithm 1 where the predicate at line 3 is replaced with Eq. 4. Then, the verification algorithm is δ-complete — i.e., if the property is not verified, it returns a δ-counterexample.*

6 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 1, and it uses the BayesOpt library [35] 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 [24]. 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 [11] 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 4.1, 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 x_* to the optimization problem
- the value of the objective function $\mathcal{F}$ (Eq. 2) at x_*
- the magnitude of the gradient of the network at x_*
- average length of the input space along each dimension

Selection. Recall from Section 4 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* [54] 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 x_* of Eq. 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 x_*. Finally, if the splitting plane is at the boundary of I, it is offset slightly so that the strategy satisfies Assumption 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.

7 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?
(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 [30] 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 [30] and CIFAR [27] datasets.

7.1 Comparison with AI^2 (RQ1)

For each network, we attempt to verify around 100 robustness properties. Following prior work [14], the evaluated robustness properties are so-called *brightening attacks* [41]. For an input point x and a threshold τ, a brightening attack consists of the input region

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

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.

Set-up. 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 [7], a state-of-the-art tool for verifying network robustness [14]. As discussed in Section 2, 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 [14], 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 6. 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.

The details for each network are shown in Figures 7 - 13. 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.

[7] Because we did not have access to the original AI^2, we reimplemented it. However, to allow for a fair comparison, we use the same underlying abstract interpretation library, and we implement the transformers exactly as described in [14].

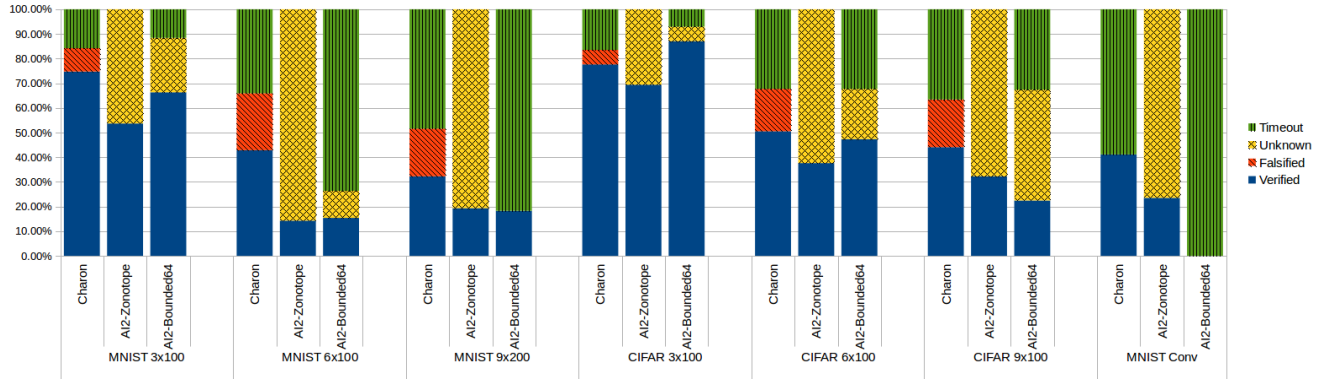

Figure 6. Summary of results for AI^2 and Charon.

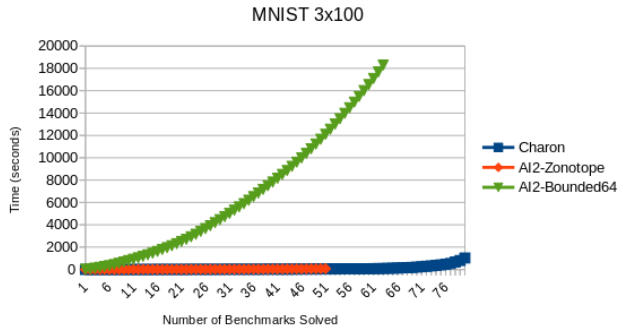

Figure 7. Comparison on a 3x100 MNIST network.

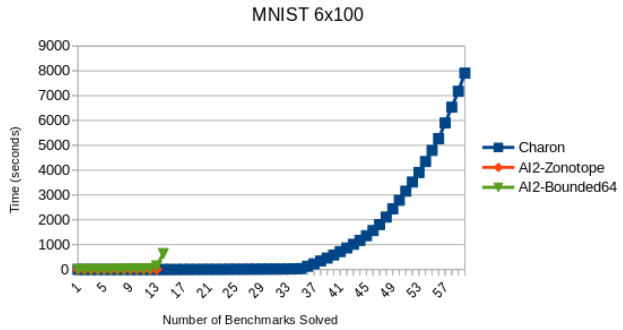

Figure 8. Comparison on a 6x100 MNIST network.

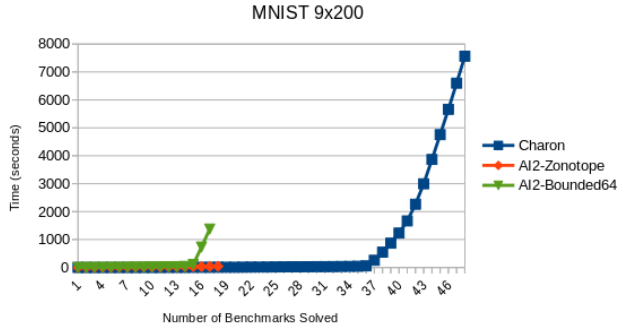

Figure 9. Comparison on a 9x200 MNIST network.

Since AI^2-Bounded64 times out on every benchmark for the convolutional network, it does not appear in Figure 13.

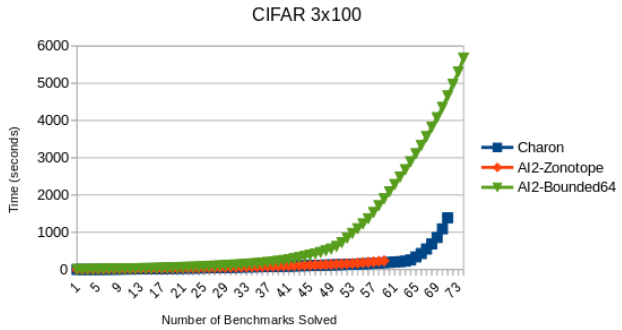

Figure 10. Comparison on a 3x100 CIFAR network.

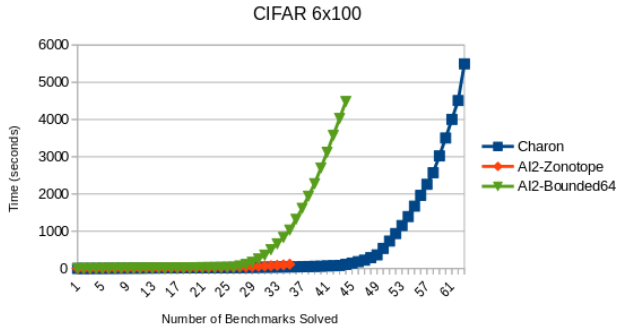

Figure 11. Comparison on a 6x100 CIFAR network.

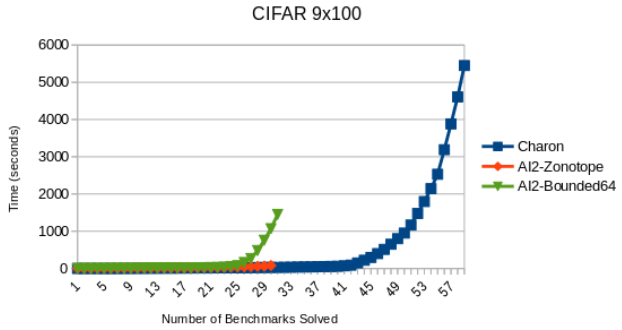

Figure 12. Comparison on a 9x100 CIFAR network.

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

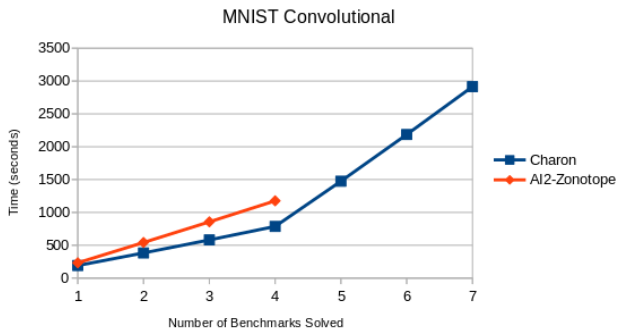

Figure 13. Comparison on a convolutional network.

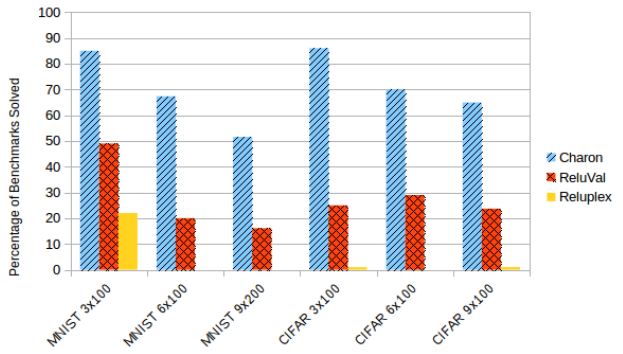

Figure 14. Comparison with ReluVal.

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 advantages of our approach compared to AI^2.

7.2 Comparison with Complete Tools (RQ1)

In this section we compare Charon with other complete tools for robustness analysis, namely ReluVal [54] and Reluplex [25]. Among these tools, Reluplex implements a variant of Simplex with built-in support for the ReLU activation function [25], 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 7.1. However, since ReluVal and Reluplex do not support convolutional layers, we exclude the convolutional net from this evaluation.

The results of this comparison are summarized in Figure 14. 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.

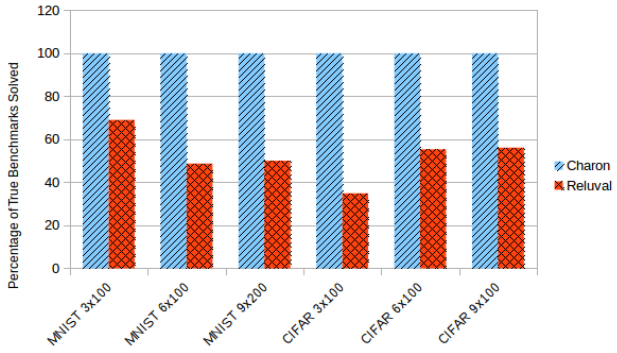

Figure 15. Comparison with ReluVal on verified benchmarks.

7.3 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 7.2, 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.

7.4 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, handcrafted strategy rather than one that is learned from data. Thus, comparing against ReluVal on the verifiably-robust benchmarks allows us to evaluate the benefits of learning a verification policy from data. [8]

The results of this comparison are shown in Figure 15. As we can see from this figure, ReluVal is still only able to solve between 35-70% 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.

[8] 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.

8 Related Work

In this section, we survey existing work on robustness analysis of neural networks and other ideas related to this paper.

Adversarial Examples and Robustness. Szegedy et al. [51] 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 [36] and malware classication [19].

Bastani et al. [4] 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 [18, 25].

Many recent papers have studied the construction of adversarial counterexamples [17, 20, 29, 33, 34, 38, 45, 52]. These approaches are based on various forms of gradient-based optimization, for example L-BFGS [50], FGSM [17] and PGD [34]. While our implementation 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. [46] used bounded model checking to verify safety of neural networks. Katz et al.[26] developed the Reluplex decision procedure extending the Simplex algorithm to verify robustness and safety properties of feedforward networks with ReLU units. Huang et al.[23] 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 [8, 32, 53] 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 [43] — in fact, like our method, 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. [14] 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 [54], 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 7, 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 [13, 22, 31, 47, 48] 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 [31, 39]. The most relevant work in this space is by Oh et al. [39], 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, our analysis of neural networks handles real-valued data and a possibly infinite space of strategies.

9 Conclusion and Future Work

We have presented a novel technique for verifying robustness properties of neural networks based on synergistically combining proof search with counterexample search. This technique makes use of black-box optimization techniques in order to learn good refinement strategies in a data-driven way. We implemented our technique and showed that it significantly outperforms state-of-the-art techniques for robustness verification. Specifically, our technique is able to solve 2.6× as many benchmarks as ReluVal and 16.6× as many as Reluplex. Our technique is able to solve more benchmarks in general than AI^2, and is able to solve them far faster. Moreover, we provide theoretical guarantees about the termination and (δ-)completeness of our approach.

In order to improve our tool in the future, we plan to explore a broader set of abstract domains and different black-box optimization techniques. Notably, one can view solver-based techniques as a perfectly precise abstract domain. While solver-based techniques have so far proven to be slow on many benchmarks, our method could learn when it is best to apply solvers and when to choose a less precise domain. This would allow the tool to combine solvers and traditional numerical domains in the most efficient way. Additionally, while Bayesian optimization fits our current framework well, it may be possible to modify the framework to work with different learning techniques which can explore higher-dimensional strategy spaces. In particular, reinforcement learning may be an interesting approach to explore in future work.

Acknowledgments

We thank our shepherd Michael Pradel as well as our anonymous reviewers and members of the UToPiA group for their helpful feedback. This material is based upon work supported by the National Science Foundation under Grants No. CCF-1162076, No. CCF-1704883, No. CNS-1646522, and No. CCF-1453386.

References

[1] [n. d.]. ELINA: ETH Library for Numerical Analysis. http://elina.ethz.ch

[2] [n. d.]. Google Cloud Platform (GCP). https://cloud.google.com/. Accessed: 2018-11-14.

[3] ApolloAuto. 2017. apollo. https://github.com/ApolloAuto/apollo

[4] Osbert Bastani, Yani Ioannou, Leonidas Lampropoulos, Dimitrios Vytiniotis, Aditya V. Nori, and Antonio Criminisi. 2016. 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*. 2613–2621.

[5] 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. 2016. End to End Learning for Self-Driving Cars. *CoRR* abs/1604.07316 (2016). http://arxiv.org/abs/1604.07316

[6] Eric Brochu, Vlad M. Cora, and Nando De Freitas. 2010. A Tutorial on Bayesian Optimization of Expensive Cost Functions, with Application to Active User Modeling and Hierarchical Reinforcement Learning. abs/1012.2599 (12 2010).

[7] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '77)*. ACM, New York, NY, USA, 238–252. https://doi.org/10.1145/512950.512973

[8] Souradeep Dutta, Susmit Jha, Sriram Sankaranarayanan, and Ashish Tiwari. 2017. Output Range Analysis for Deep Neural Networks. *CoRR* abs/1709.09130 (2017). arXiv:1709.09130 http://arxiv.org/abs/1709.09130

[9] Andre Esteva, Brett Kuprel, Roberto A. Novoa, Justin Ko, Susan M. Swetter, Helen M. Blau, and Sebastian Thrun. 2017. Dermatologist-level classification of skin cancer with deep neural networks. *Nature* 542 (01 2017), 115–118. http://dx.doi.org/10.1038/nature21056

[10] Ivan Evtimov, Kevin Eykholt, Earlence Fernandes, Tadayoshi Kohno, Bo Li, Atul Prakash, Amir Rahmati, and Dawn Song. 2017. Robust Physical-World Attacks on Machine Learning Models. *CoRR* abs/1707.08945 (2017). arXiv:1707.08945 http://arxiv.org/abs/1707.08945

[11] Message P Forum. 1994. *MPI: A Message-Passing Interface Standard*. Technical Report. Knoxville, TN, USA.

[12] Sicun Gao, Jeremy Avigad, and Edmund M. Clarke. 2012. δ-complete Decision Procedures for Satisfiability over the Reals. In *Proceedings of the 6th International Joint Conference on Automated Reasoning (IJCAR'12)*. Springer-Verlag, Berlin, Heidelberg, 286–300. https://doi.org/10.1007/978-3-642-31365-3_23

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

[14] Timon Gehr, Matthew Mirman, Dana Drachsler-Cohen, Petar Tsankov, Swarat Chaudhuri, and M. Vechev. 2018. AI2: Safety and Robustness Certification of Neural Networks with Abstract Interpretation. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, San Francisco, CA, USA, 3–18. https://doi.org/10.1109/SP.2018.00058

[15] Khalil Ghorbal, Eric Goubault, and Sylvie Putot. 2009. The Zonotope Abstract Domain Taylor1+. In *Proceedings of the 21st International Conference on Computer Aided Verification (CAV '09)*. Springer-Verlag, Berlin, Heidelberg, 627–633. https://doi.org/10.1007/978-3-642-02658-4_47

[16] Yuan Gong and Christian Poellabauer. 2018. An Overview of Vulnerabilities of Voice Controlled Systems. *arXiv preprint arXiv:1803.09156* (2018).

[17] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. 2015. Explaining and Harnessing Adversarial Examples. In *International Conference on Learning Representations*.

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

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

[20] Kathrin Grosse, Nicolas Papernot, Praveen Manoharan, Michael Backes, and Patrick D. McDaniel. 2016. Adversarial Perturbations Against Deep Neural Networks for Malware Classification. *CoRR* abs/1606.04435 (2016). http://arxiv.org/abs/1606.04435

[21] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 770–778. https://doi.org/10.1109/CVPR.2016.90

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

[23] Xiaowei Huang, Marta Kwiatkowska, Sen Wang, and Min Wu. 2016. Safety Verification of Deep Neural Networks. *CoRR* abs/1610.06940 (2016).

[24] Kyle D. Julian, Jessica Lopez, Jeffrey S. Brush, Michael P. Owen, and Mykel J. Kochenderfer. 2016. Policy compression for aircraft collision avoidance systems. In *2016 IEEE/AIAA 35th Digital Avionics Systems Conference (DASC)*. 1–10. https://doi.org/10.1109/DASC.2016.7778091

[25] Guy Katz, Clark Barrett, David L. Dill, Kyle Julian, and Mykel J.Kochenderfer. 2017. Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks. In *Proceedings of the 29th International Conference On Computer Aided Verification*.

[26] Guy Katz, Clark W. Barrett, David L. Dill, Kyle Julian, and Mykel J. Kochenderfer. 2017. Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks. *CoRR* abs/1702.01135 (2017).

[27] Alex Krizhevsky. 2009. *Learning Multiple Layers of Features from Tiny Images*. Technical Report.

[28] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger (Eds.). Curran Associates, Inc., 1097–1105. http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

[29] Alexey Kurakin, Ian J. Goodfellow, and Samy Bengio. 2016. Adversarial Machine Learning at Scale. *CoRR* abs/1611.01236 (2016). arXiv:1611.01236 http://arxiv.org/abs/1611.01236

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

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

[32] Alessio Lomuscio and Lalit Maganti. 2017. An approach to reachability analysis for feed-forward ReLU neural networks. *CoRR* abs/1706.07351 (2017). arXiv:1706.07351 http://arxiv.org/abs/1706.07351

[33] Chunchuan Lyu, Kaizhu Huang, and Hai-Ning Liang. 2015. 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*. 301–309.

[34] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. 2018. Towards deep learning models resistant to adversarial attacks.

[35] Ruben Martinez-Cantin. 2014. BayesOpt: A Bayesian Optimization Library for Nonlinear Optimization, Experimental Design and Bandits. *Journal of Machine Learning Research* 15 (2014), 3915–3919. http://jmlr.org/papers/v15/martinezcantin14a.html

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

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

[38] Anh Mai Nguyen, Jason Yosinski, and Jeff Clune. 2015. 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*. 427–436.

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

[40] Nicolas Papernot, Patrick D. McDaniel, and Ian J. Goodfellow. 2016. Transferability in Machine Learning: from Phenomena to Black-Box Attacks using Adversarial Samples. *CoRR* abs/1605.07277 (2016). arXiv:1605.07277 http://arxiv.org/abs/1605.07277

[41] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. 2017. DeepXplore: Automated Whitebox Testing of Deep Learning Systems. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. ACM, New York, NY, USA, 1–18. http://doi.acm.org/10.1145/3132747.3132785

[42] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. 2017. Towards Practical Verification of Machine Learning: The Case of Computer Vision Systems. *CoRR* abs/1712.01785 (2017). arXiv:1712.01785 http://arxiv.org/abs/1712.01785

[43] Luca Pulina and Armando Tacchella. 2010. 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*. 243–257.

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

[45] Sara Sabour, Yanshuai Cao, Fartash Faghri, and David J. Fleet. 2015. Adversarial Manipulation of Deep Representations. *CoRR* abs/1511.05122 (2015).

[46] Karsten Scheibler, Leonore Winterer, Ralf Wimmer, and Bernd Becker. 2015. 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*. 30–40.

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

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

[49] Gagandeep Singh, Markus Püschel, and Martin Vechev. 2017. Fast Polyhedra Abstract Domain. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. ACM, New York, NY, USA, 46–59. https://doi.org/10.1145/3009837.3009885

[50] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. 2013. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199* (2013).

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

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

[53] Vincent Tjeng and Russ Tedrake. 2017. Verifying Neural Networks with Mixed Integer Programming. *CoRR* abs/1711.07356 (2017). arXiv:1711.07356 http://arxiv.org/abs/1711.07356

[54] Shiqi Wang, Kexin Pei, Justin Whitehouse, Junfeng Yang, and Suman Jana. 2018. Formal Security Analysis of Neural Networks using Symbolic Intervals. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 1599–1614. https://www.usenix.org/conference/usenixsecurity18/presentation/wang-shiqi

[55] 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. 2016. Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation. *CoRR* abs/1609.08144 (2016). http://arxiv.org/abs/1609.08144

[56] Xiaoyong Yuan, Pan He, Qile Zhu, Rajendra Rana Bhat, and Xiaolin Li. 2017. Adversarial Examples: Attacks and Defenses for Deep Learning. *CoRR* abs/1712.07107 (2017). arXiv:1712.07107 http://arxiv.org/abs/1712.07107

[57] Zhenlong Yuan, Yongqiang Lu, Zhaoguo Wang, and Yibo Xue. 2014. Droid-Sec: Deep Learning in Android Malware Detection. In *Proceedings of the 2014 ACM Conference on SIGCOMM (SIGCOMM '14)*. ACM, 371–372. http://doi.acm.org/10.1145/2619239.2631434

[58] Zhenlong Yuan, Yongqiang Lu, and Yibo Xue. 2016. Droiddetector: android malware characterization and detection using deep learning. *Tsinghua Science and Technology* 21, 1 (Feb 2016), 114–123. https://doi.org/10.1109/TST.2016.7399288

A Proofs

In this section we present the proofs of the theorems in Section 5. For convenience, the assumptions and theorem statements have been copied.

Definition 5.3. *For a given network $\mathcal{N}$, input region I, target class M, and $\delta > 0$, a δ-counterexample is a point $x \in I$ such that for some j with $1 \leq j \leq m$ and $j \neq M$, $\mathcal{N}(x)_M - \mathcal{N}(x)_j \leq \delta$.*

Definition 5.1. *For any set $X \subseteq \mathbb{R}^n$, its* diameter *$D(X)$ is defined as*

$$D(X) = \sup\{\|x_1 - x_2\|_2 \mid x_1, x_2 \in X\}$$

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

Assumption 1. *There exists some $\lambda \in \mathbb{R}$ with $0 < \lambda < 1$ such that for any network $\mathcal{N}$, input region I, and point $x_* \in I$, if $(I_1, I_2) = \textsc{Refine}(N, I, x_*)$, then $D(I_1) < \lambda D(I)$ and $D(I_2) < \lambda D(I)$.*

Assumption 2. *Let $\mathcal{N}^\#$ be the abstract transformer representing a network $\mathcal{N}$. Let a be an element of the abstract domain representing the input region I. We assume there exists some $K_\mathcal{N} \in \mathbb{R}$ such that $D(\gamma(\mathcal{N}^\#(\alpha(I)))) < K_\mathcal{N} D(I)$.*

Theorem 5.2. *Consider the variant of Algorithm 1 where the predicate at line 3 is replaced with Eq. 4. Then, the verification algorithm always terminates under Assumptions 1 and 2.*

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

By Assumption 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, 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 1 must terminate without recurring. In this case, suppose x_* is the point returned by the call to Minimize and the algorithm does not terminate. Then $\mathcal{F}(x_*) > \delta$ and in particular, $\mathcal{N}(x_*)_K - \mathcal{N}(x_*)_i > \delta$. Since Analyze is sound, we must have $\mathcal{N}(x_*) \in F(I_k)$. Then since $D(F(I_k)) < \delta/2$, we must have that for any point $y' \in F(I_k)$, $\|\mathcal{N}(x_*) - y'\|_2 < \delta/2$. In particular, for all i, $|(\mathcal{N}(x_*))_i - y'_i| < \delta/2$, so $y'_i > (\mathcal{N}(x_*))_i - \delta/2$ and $y'_i < (\mathcal{N}(x_*))_i + \delta/2$. Then, for all i,

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

Thus, for each point $y' \in F(I_k)$, we have $y'_K > y'_i$. Since 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 1 is bounded, so it must terminate. □

Theorem 5.4. *Consider the variant of Algorithm 1 where the predicate at line 3 is replaced with Eq. 4. Then, the verification algorithm is δ-complete, meaning that if the algorithm does not return "Verified" then the return value is a δ-counterexample for the property.*

Proof. First note that by Theorem 5.2, Algorithm 1 must terminate. Therefore the algorithm must return some value, and we can prove this theorem by analyzing the possible return values. There are five places at which Algorithm 1 can terminate: lines 4, 8, 12, 15, and 16. We only care about the case where the algorithm does not return "Verified" so we can ignore lines 8 and 16. The return at line 4 is only reached after checking that x_* is a δ-counterexample, so clearly if that return statement is used then the algorithm returns a δ-counterexample. This leaves the return statements at lines 12 and 15. We suppose by induction that the recursive calls at lines 10 and 13 are δ-complete. Then if r_1 is not Verified, it must be a δ-counterexample. Thus the return statement at line 12 also returns a δ-counterexample. Similarly, if r_2 is not Verified, then it is a δ-counterexample, so line 15 returns a δ-counterexample. □