Deepcause: Verifying neural networks with abstraction refinement

NGUYEN HUA GIA PHUC
Singapore Management University, hgpnguyen.2019@phdcs.smu.edu.sg

DEEPCAUSE: VERIFYING NEURAL NETWORKS WITH ABSTRACTION REFINEMENT

NGUYEN HUA GIA PHUC

SINGAPORE MANAGEMENT UNIVERSITY

2022

DeepCause: Verifying Neural Networks with Abstraction Refinement

Nguyen Hua Gia Phuc

Submitted to School of Computing and Information Systems
in partial fulfillment for the requirements of the Degree of

Master of Philosophy in Information Systems

SINGAPORE MANAGEMENT UNIVERSITY

2022

Declaration Page

I hereby declare that this thesis is my original work and it has been written by me in its entirety.
I have duly acknowledged all the sources of information which have been used in this thesis

This thesis has also not been submitted for any degree in any university previously

[illegible]

Nguyen Hua Gia Phuc
25 July 2022

Abstract

Neural networks have been becoming essential parts in many safety-critical systems (such as self-driving cars and medical diagnosis). Due to that, it is desirable that neural networks not only have high accuracy (which traditionally can be validated using a test set) but also satisfy some safety properties (such as robustness, fairness, or free of backdoor). To verify neural networks against desired safety properties, there are many approaches developed based on classical abstract interpretation. However, like in program verification, these approaches suffer from false alarms, which may hinder the deployment of the networks.

One natural remedy to tackle the problem adopted from program verification community is counterexample-guided abstraction refinement (*CEGAR*). The application of *CEGAR* in neural network verification is, however, highly non-trivial due to the complication raised from both neural networks and abstractions. In this thesis, we propose a method to enhance abstract interpretation in verifying neural networks through an application of *CEGAR* in two steps. First, we employ an optimization-based procedure to validate abstractions along each propagation step and identify problematic abstraction via counterexample searching. Then, we leverage a causality approach to select the most likely problematic components of the abstraction and refine them accordingly.

To evaluate our approach, we have implemented a prototype named *DeepCause* based on *DeepPoly* and take local robustness as the target safety property to verify. The evaluation shows that our proposal can outperform *DeepPoly* and *RefinePoly* in all benchmark networks. We should note that our idea is not limited to specific abstract domain and we believe it is a promising step towards enhancing verification of complex neural network systems.

Contents

Acknowledgement

I would like to acknowledge and give my thank to my supervisor professor Sun Jun, who make this work possible. I would also like to thank my senior Pham Hong Long who has guided and given me advice through all stage of my project.

Finally, I would like to give special thanks to my family as a whole for their continuous support and understand when undertaking my research project.

Chapter 1

Introduction

Recently, neural networks have become essential parts in many safety-critical systems such as malware detection [42], self-driving cars [4], and medical diagnosis [19]. Due to this reason, besides the high accuracy requirement, it is thus desirable to formally verify such neural networks against safety properties, especially before their deployment into real scenarios.

Inspired by decision making procedure from human beings, the community has identify and formalized several interesting properties for neural networks, such as local robustness and fairness. Local robustness means that the networks output remains consistent even under small perturbation on an input and a network is fair if it's classification is not affected by the different values of the chosen feature. In this thesis, we focus on the neural network verification problem with respect to local robustness property, i.e, given a specific input, a perturbation within certain range to the input does not change its label. More formally, a neural network N is local robust on an input x with perturbing range r if and only if $\forall x' \cdot ||x - x'||_p < r \implies N(x') = N(x)$ (assuming $L_p - norm$ is used).

To tackle the above problem, many approaches are developed based on classical abstract interpretation idea from program verification, such as *AI^2* [13], *ReluVal* [39], *DeepZ* [28], *DeepPoly* [30], *RefineZono* [29], and *RefinePoly* [27]. Unfortunately, like in program verification based on over-approximation, these approaches may suffer from false alarms, which may hinder the deployment of the networks.

One intuitive way to enhance existing abstract interpretation is to partition the input region. With smaller input region, the precision loss of abstract interpretation usually decreases. To the best of our knowledge, there is one approach adopts such idea, i.e., *ReluVal* [39] bisects the input space with the guidance of symbolic interval gradients. However, this method concretizes the symbolic interval and ignore their dependencies at ReLU layer which causes over-approximate.

In this work, we aim to enhance existing abstract interpretation via counterexample-guided abstraction refinement (*CEGAR*). *CEGAR* has been proven effective for program verification [1]. Applying *CEGAR* to neural network verification is however non-trivial due to the complex nature of neural networks and complicated numerical abstractions. Once we fail to verify a property based on an abstraction of the given neural network, the main challenges are how to identify problematic abstraction effectively and how to refine the abstraction accordingly.

In the case of program verification, given a counterexample in the form of an abstract program path, symbolic execution is typically applied to check whether the counterexample is spurious and methods such as interpolation [25] or weakest precondition computation [8] can be applied to answer the questions. This is however infeasible in the case of neural networks since neurons are often fully connected and there is no single path that can be identified for symbolic execution, even if we overlook the cost of symbolic execution on neural networks.

Furthermore, in the classic *CEGAR*, we construct an abstraction of the system and verify it after the abstraction is complete. If the verification fails, the abstraction is refined based on the counterexample, i.e., by eliminating certain problematic over-approximation. Sometimes many abstraction refinement iterations are necessary. This is likely the case in verifying neural networks, as over-approximation happens for every neuron. Furthermore, as neural networks are often fully connected, a problematic over-approximation in a layer is likely to propagate to subsequent layers. It is thus desirable to identify problematic over-approximation early.

Our answer to these questions is as follows. Instead of collecting counterexamples after abstract interpretation, we validate abstraction at each propagation step. In this way, we could save the cost of symbolic execution and have the chance identify problematic abstraction early. In specific, we formulate an optimization problem to search for counterexamples. Note that neural network is designed for optimization and as a result, this optimization problem can often be solved effectively using optimization techniques associated with neural networks.

After that, with the root problematic over-approximation, we try to eliminate the counterexample through partitioning the very last abstraction. Rather than partition the abstraction of all neurons which would result in an exponential number of new abstractions, we choose to select the top k problematic neurons and only partition these neurons in order to efficiently refine the abstraction. There are several ways to rank the neurons. One of the most popular method is gradient, many papers like [[39], [38]] use gradient to determine the attribution of each neuron. Attributions are defined as the effect of an input feature on the prediction function's output [34] which means that it show how much impact a neuron has on the outcome. However, gradient method are shown to be sensitive to even a simple constant shift in input vector [18]. As a result, we use new attribution method for neural networks developed using first principles of causality [7] to select the top k neurons. This method not only doesn't have the same drawback but it also proved to capture the causal influence of neuron and products better result than state-of-the-art gradient-based method [7].

To evaluate this idea, we have implemented our proposal in a prototype named *DeepCause*. In specific, *DeepCause* is based on *DeepPoly* which is the state-of-the-art abstract interpretation neural network analyzer. We took a set of 45 neural network models trained on the MNIST dataset with different activation functions and evaluated their robustness bounds for the first 100 test inputs. The results show that our approach can effectively improve the performance of *DeepPoly*. Moreover, as equipped with an optimization-based validation procedure, *DeepCause* can also effectively reject non-robust bounds for specific inputs. In addition, as the idea is quite general and thus applicable to other abstract domains than *DeepPoly*, we believe *DeepCause* provides a promising step towards enhancing the verification of complex systems.

Organization The rest of this paper is organized as follows. Section 2 demonstrates our approach through an example. Section 3 reviews preliminary knowledge. Section 4 presents our approach in detail. In Section 5, we show the implementation detail and evaluation results. Section 6 reviews related work. Section 7 concludes and proposes potential research directions.

Chapter 2

An Illustrative Example

In this section, we illustrate how our proposal works through a simple example. For simplicity, we write I to be an input to a neural network model and I_j to be its j-th feature value. Similarly, we write O to be the output of the model and O_j to be its j-th feature value.

Example 2.1 (A Simple Example)**.** Let us consider the simple fully-connected feed-forward neural network shown in Fig. 2.1. It consists of four layers, i.e., an input layer, two hidden layers, and an output layer. Each layer has two neurons. The input layer reads the input; each hidden layer processes its inputs via an affine transformation and a *ReLU* activation function; and the output layer yields an output vector. A label is then generated based on the largest value in the output vector. The weights on the edges represent the coefficients of the weight matrix and the values beside each nodes are the biases for the affine transformations at each layer.

The property to be verified is local robustness. Assume we are given an input $(0, 0)$ with output label 0. The verification task is to prove that $O_0 > O_1$ for all I such that $-1 \leq I_0 \leq 1$ and $-1 \leq I_1 \leq 1$. In other words, the neural network is local robust on input $(0, 0)$ against a perturbation with $r = 1$.

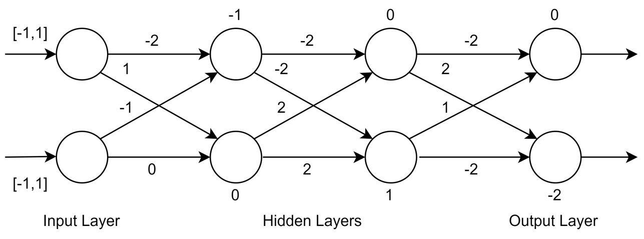

Figure 2.1: An illustrative example feed-forward *ReLU* neural network with one input layer, two hidden layers and one output layer, each layer containing two neurons.

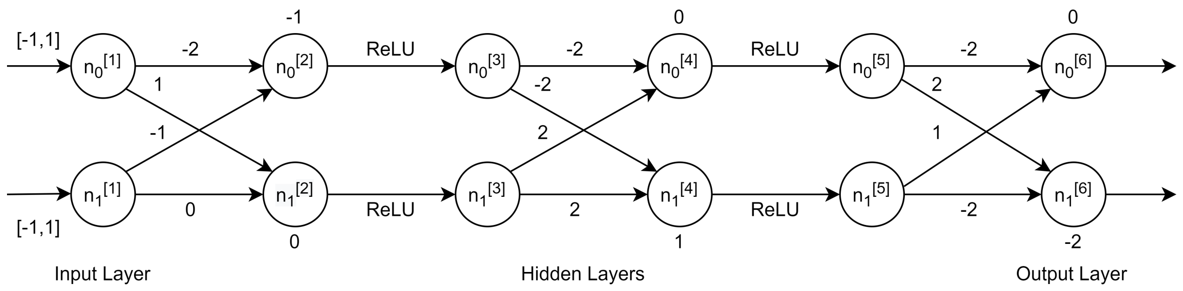

Figure 2.2: Expanded feed-forward neural network from Figure 2.1

2.1 Existing Approaches on Verifying the Example

In the following, we first show how existing abstract interpretation-based verification methods work on this example. Due to the different characteristics of affine operations and activation operations, the network is transformed by expanding each neuron into two nodes, i.e., one for the affine operation and one for the activation function, in the pre-processing step. That is, given the network shown in Fig. 2.1, the result of the transformation is shown in Fig. 2.2.

Abstract interpretation-based approaches reason about the target system's behaviors via abstractions, which are defined via abstract transformers. Different from concrete transformer f which takes an concrete state as input and yields a concrete output, abstract transformer $f^{\#}$ takes an abstract state, i.e. a set of concrete states, as input and yields another abstract state as output. To guarantee the soundness, the output abstract states must over-approximate all the reachable output concrete states from the input concrete states. The domain of the abstract states is called an abstract domain.

In practice, there are several numerical abstract domains available, and we take *DeepPoly* abstract domain as an example. Here we briefly introduce the *DeepPoly* domain and leave the details to Section 3. A *DeepPoly* abstraction a for one neuron is defined as a tupple $a = \langle a^{\leq}, a^{\geq}, l, u \rangle$, where $a^{\leq}, a^{\geq}$ are two polyhedral constraints presents symbolic lower and upper bounds; and l, u are concrete lower and upper bounds which over-approximate the two symbolic bounds.

Example 2.2 (Verifying Example 2.1 with *DeepPoly*)**.** We write $x_j^{[i]}$ where $i \in 1..6$ and $j \in \{0, 1\}$ to denote the abstraction of node $n_j^{[i]}$ in Fig. 2.2. First, after the input layer (layer 0), we transform the input range $[-1, 1] \times [-1, 1]$ into a *DeepPoly* element $x^{[1]}$ such that

$$\begin{aligned} -1 \leq x_0^{[1]} \leq 1, &\quad l_0^{[1]} = -1, u_0^{[1]} = 1 \\ -1 \leq x_1^{[1]} \leq 1, &\quad l_1^{[1]} = -1, u_1^{[1]} = 1 \end{aligned}$$

Then to compute abstraction $x^{[2]}$, we have $x^{[2]} = \mathit{Affine}^{\#}(x^{[1]})$. As the affine operation on DeepPoly is exact, we can safely write as $x^{[2]} = \begin{bmatrix} -2 & -1 \\ 1 & 0 \end{bmatrix} * x^{[1]} + \begin{bmatrix} -1 \\ 0 \end{bmatrix}$. Then we have

$$\begin{gathered} -2x_0^{[1]} - x_1^{[1]} - 1 \leq x_0^{[2]} \leq -2x_0^{[1]} - x_1^{[1]} - 1 \\ x_0^{[1]} \leq x_1^{[2]} \leq x_0^{[1]} \end{gathered}$$

where $l_0^{[2]} = -4, u_0^{[2]} = 2$ and $l_1^{[2]} = -1, u_1^{[2]} = 1$. Next, we can obtain the following abstraction for each node in the figure.

$$\begin{gathered} 0 \leq x_0^{[3]} \leq 0.33x_0^{[2]} + 1.33, \quad l_0^{[3]} = 0, u_0^{[3]} = 2 \\ 0 \leq x_1^{[3]} \leq 0.5x_1^{[2]} + 0.5, \quad l_1^{[3]} = 0, u_1^{[3]} = 1 \\ \ldots \\ -2x_0^{[5]} + x_1^{[5]} \leq x_0^{[6]} \leq -2x_0^{[5]} + x_1^{[5]}, \quad l_0^{[6]} = -1.67, u_0^{[6]} = 5 \\ 2x_0^{[5]} - 2x_1^{[5]} - 2 \leq x_1^{[6]} \leq 2x_0^{[5]} - 2x_1^{[5]} - 2, \; l_1^{[6]} = -12, u_1^{[6]} = -1.33 \end{gathered}$$

Lastly, to verify the property, we check whether the expression

$$x_0^{[6]} - x_1^{[6]} \geq 0$$

is always true, i.e., the lower bound of the left-hand side is larger than 0. If it is, the property $O_0 > O_1$ is verified. Unfortunately, the lower bound of the left-hand side is -0.33, which is smaller than 0. As a result, *DeepPoly* fails to prove the property. Note that this does not indicate the neural network is not robust within the given input range. □

It is not uncommon that existing abstract interpretation-based methods fail to verify a property. For the above example, *DeepZ* which uses a different abstract domain, similarly fails to verify the property. This is because spurious counterexamples introduced by over-approximation prevent the property from being verified.

One way to address this issue is to refine the abstraction, i.e., to identify an alternative abstraction based on which the property can be verified. The space of abstractions is however huge. Because abstraction is applied to every neuron, if there are k ways of refining a neuron, the number of possible abstractions would be k^n where n is the number of neurons. Given that the number of neurons in real-world neural networks is often large, ranges from hundreds to millions, we need a way of identifying neurons with likely problematic abstraction.

2.2 An Overview of Our Approach

Our approach adopts *CEGAR* to solve the issue of problematic over-approximation and further improves the classic *CEGAR* based on the characteristics of neural networks. In the following, we show how our approach works to verify the example. The details of our approach are presented in Section 4.

Example 2.3. We start with formulating a constrained optimization problem to check whether there exists some value I which falsifies the property as follows.

$$\begin{aligned} \text{minimize} \quad & O_0 - O_1 \\ \text{subject to} \quad & -1 \leq I_0 \leq 1 \quad \wedge \quad -1 \leq I_1 \leq 1 \end{aligned}$$

Note that solving this optimization problem is straightforward based on gradient descent since neural networks are designed for optimization. If the optimization successfully identifies an input such that $O_0 - O_1 \leq 0$, a counterexample is identified and thus the property is falsified. In this example, we fail to identify such an input with thousands of iterations, which is taken as a hint that the property might be valid.

Next, we compute the abstractions $x^{[1]}$ and $x^{[2]}$ as shown in Example 2.2. Recall that $x^{[3]}$ is over-approximations. We then formulate a second optimization problem to check whether there exists some concrete value of $x^{[3]}$ which satisfies the lower bound, upper bound as well as $O_0 \leq O_1$, i.e., to check whether the over-approximation introduced by

$x^{[3]}$ is problematic. In detail, an optimization problem similar to the one above (i.e., the bounds of $x^{[3]}$ are used instead of $x^{[1]}$) is formulated and solved to find a counterexample at layer 3. Unfortunately, we cannot find any counterexample at this layer either.

However, at layer 5, within several to dozens of iterations, the optimization procedure successfully identifies a 'counterexample', e.g., $x^{[5]} = (2, 1)$ with the output vector $(-3, 0)$. Either it is a part of some real counterexamples (that we missed in solving the first optimization problem) or part of a spurious counterexample introduced by $x^{[5]}$ (in which case refining $x^{[5]}$ would eliminate it). In our approach, we assume the latter and only examine the possibility of the former if the latter is proven not the case.

We thus refine $x^{[5]}$ next. In this example, the activation function is *ReLU* and thus refinement can be done by simply distinguishing whether the input to the *ReLU* function is positive or not. Out of the two features of $x^{[5]}$, we choose $x_0^{[5]}$ as $x_1^{[5]}$ is precise. In a more general case, we may choose the feature based on the causal analysis. The Average Causal Effect (ACE) measure causal strength of various input features towards the label output neuron (in this case $x_0^{[6]}$). It is calculated by computing the impact of the input feature ($x_0^{[5]}$ and $x_1^{[5]}$) with fixed value β on the output feature ($x_0^{[6]}$). The value β is taken uniformly from the neuron's interval ($l_0^{[5]} = 0, u_0^{[5]} = 0; l_1^{[5]} = 1, l_1^{[5]} = 5$) and the ACE calculate the average attribution of all the β. Intuitively, a feature with a larger causality on the output is more likely the problematic contribution and thus refining the abstraction on the feature is more likely to eliminate the counterexamples. More details about this causality-based selection is shown in Chapter 4.

To refine $x_0^{[5]}$, we split its input $x_0^{[4]}$ into two parts $x_0^{[4,p0]}$ and $x_0^{[4,p1]}$ at point 0:

$$x^{[4,p0]} : \begin{cases} -2x_0^{[3]} + 2x_1^{[3]} \le x_0^{[4,p0]} \le -2x_0^{[3]} + 2x_1^{[3]} \\ -2x_0^{[3]} + 2x_1^{[3]} + 1 \le x_1^{[4,p0]} \le -2x_0^{[3]} + 2x_1^{[3]} + 1 \\ l_0^{[4,p0]} = -4, u_0^{[4,p0]} = 0; l_1^{[4,p0]} = 1, u_1^{[4,p0]} = 5 \end{cases}$$

$$x^{[4,p1]} : \begin{cases} -2x_0^{[3]} + 2x_1^{[3]} \le x_0^{[4,p1]} \le -2x_0^{[3]} + 2x_1^{[3]} \\ -2x_0^{[3]} + 2x_1^{[3]} + 1 \le x_1^{[4,p1]} \le -2x_0^{[3]} + 2x_1^{[3]} + 1 \\ l_0^{[4,p1]} = 0, u_0^{[4,p1]} = 2; l_1^{[4,p1]} = 1, u_1^{[4,p1]} = 5 \end{cases}$$

For the abstraction part where $x_0^{[4]} < 0$, we have

$$0 \le x_0^{[5,p0]} \le 0, \quad l_0^{[5,p0]} = 0, u_0^{[5,p0]} = 0$$
$$x_1^{[4,p0]} \le x_1^{[5,p0]} \le x_1^{[4,p0]}, \quad l_1^{[5,p0]} = 1, u_0^{[5,p0]} = 5$$

and until the output layer, we have

$$
\begin{gathered}
-2x_0^{[5,p0]} + x_1^{[5,p0]} \le x_0^{[6,p0]} \le -2x_0^{[5,p0]} + x_1^{[5,p0]} \\
2x_0^{[5,p0]} - 2x_1^{[5,p0]} - 2 \le x_1^{[6,p0]} \le 2x_0^{[5,p0]} - 2x_1^{[5,p0]} - 2 \\
l_0^{[6,p0]} = 1, u_0^{[6,p0]} = 5; \quad l_1^{[6,p0]} = -12, u_1^{[6,p0]} = -4
\end{gathered}
$$

and apparently, it clears from the bound that we have $x_0^{[6,p0]} > x_1^{[6,p0]}$. For the other case where abstraction is $x^{[4,p1]}$, we do the same

$$
\begin{gathered}
x_0^{[4,p1]} \le x_0^{[5,p1]} \le x_0^{[4,p1]}, \quad l_0^{[5,p1]} = 0, u_0^{[5,p1]} = 2 \\
x_1^{[4,p1]} \le x_1^{[5,p01} \le x_1^{[4,p1]}, \quad l_1^{[5,p1]} = 1, u_0^{[5,p1]} = 5
\end{gathered}
$$

and

$$
\begin{gathered}
-2x_0^{[5,p1]} + x_1^{[5,p1]} \le x_0^{[6,p1]} \le -2x_0^{[5,p1]} + x_1^{[5,p1]} \\
2x_0^{[5,p1]} - 2x_1^{[5,p1} - 2 \le x_1^{[6,p1]} \le 2x_0^{[5,p1]} - 2x_1^{[5,p1]} - 2 \\
l_0^{[6,p1]} = -1, u_0^{[6,p1]} = 5; \quad l_1^{[6,p1]} = -12, u_1^{[6,p1]} = -4
\end{gathered}
$$

which also results in $x_0^{[6,p1]} > x_1^{[6,p1]}$. Therefore, the label should be 0.

From a high level view, although the original problem can not be verified using *DeepPoly*, our approach can verify it through first splitting it into two sub-problems, and then verifying them separately. In fact, it is a crux to figure out which abstraction to refine and how to refine the abstraction, both of which will be detailed in the rest of this paper.

Chapter 3

Preliminaries

In this section, we provide the necessary background on neural networks and abstract interpretation. Further, we show how abstract interpretation is used to verify neural networks in existing approaches.

3.1 Neural Networks

In general, a neural network can be viewed as a function $N : \mathbb{R}^p \rightarrow \mathbb{R}^q$ mapping an input $x \in \mathbb{R}^p$ (e.g., images or texts) to an output $y \in \mathbb{R}^q$ (e.g., labels for image classification or texts for machine translation). In this work, we focus on deep feed-forward neural networks that can be organized using a layered architecture, where data flows from layer to layer.

The first layer is the input layer; the last layer is the output layer and the remaining layers are hidden layers. Based on operation a layer performs, there are two common used layers: affine layer and activation layer. An affine layer applies an affine operation i.e., $\mathit{Affine}(x) = Ax + b$ where A is a weight matrix and b is a bias vector. An activation layer applies a non-linear activation function σ. Widely used activation functions are *ReLU*, *Sigmoid*, and *Tanh*. Activation function is applied neuron-wise, e.g., given an input $x = (x_0, \cdots, x_{p-1}) \in \mathbb{R}^p$, $\mathit{ReLU}(x) = \big(\mathit{ReLU}(x_0), \cdots, \mathit{ReLU}(x_{p-1})\big)$.

In the following, we assume a neural network N consists of k layers, and each layer i

contains d_i neurons. Then layer i is a function $f_i : \mathbb{R}^{d_i} \to \mathbb{R}^{d_{i+1}}$ mapping the input of layer i, i.e., $x^{[i]}$, to the input of layer $i+1$, i.e., $x^{[i+1]}$, and the neural network is function $N : \mathbb{R}^{d_0} \to \mathbb{R}^{d_k}$, where d_k is the dimension of output. Usually the input layer only reads in data and does not transform the data, and thus we can simply write $x^{[1]} = f_0(x^{[0]}) = x^{[0]}$.

3.2 Problem Definition

A verification task in our setting is a triple $\{\phi_I\}N\{\phi_O\}$ where ϕ_I is a constraint on the inputs; $N = f_k \circ f_{k-1} \circ \cdots f_i \circ \cdots \circ f_0$ is a feed-forward neural network where each f_i is a layer of neurons as discussed in Section 3.1; and ϕ_O is a constraint on the outputs. Given N, we write N_i to denote $f_k \circ f_{k-1} \circ \cdots \circ f_i$, i.e., a partial neural network where the i-layer becomes the input layer.

The triple is invalid if there exists an input $x^{[0]}$ such that $x^{[0]} \models \phi_I$ and $N(x^{[0]}) \models \neg\phi_O$. The input $x^{[0]}$ is referred to as a counterexample. Otherwise, the triple is valid. A counterexample for N_i where $i > 0$ is referred to as a partial counterexample, i.e., a valuation of input to the i-th layer which violates ϕ_O. The verification problem is defined as follows. *Given a triple $\{\phi_I\}N\{\phi_O\}$, check whether the triple is valid or not.*

In this work, the primary means of verifying neural network is abstraction. Given a mapping f, we say that a relation F is an abstraction of f if and only if $\forall i,\ f(i) = o$, such that $(i, o) \in F$ (i.e., F over-approximates f) and F is less complicated than f in some measure. An abstraction is 'good' if the property can be effectively verified based on the abstraction. However, it should be noted that abstraction may introduce spurious counterexamples, in which case we say that the over-approximation is problematic.

3.3 Abstract Interpretation for Neural Networks

Abstract interpretation is a classic static program analysis technique. The idea is to soundly approximate and reason about a program' behaviors without executing it. Abstract interpretation concerns two domains, i.e., a concrete domain $\mathcal{C}$ where program states are defined, and an abstract domain $\mathcal{A}$ where abstractions are defined.

In abstract interpretation, the choice of abstract domain is crucial as it determines the

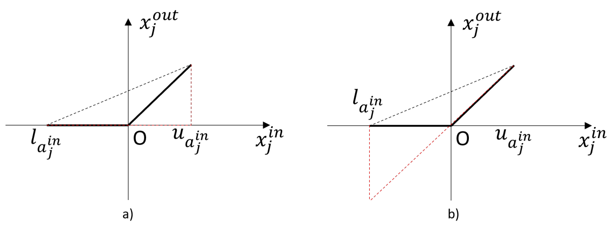

Figure 3.1: Convex approximations for *ReLU* in *DeepPoly*, when $l_j^{in} < 0$ and $l_j^{in} > 0$ is satisfied.

efficiency and precision of the abstract interpretation. In the literature, several numerical abstract domains are available for analyzing neural networks, such as interval, zonotope, and polyhedral. We focus on the *DeepPoly* abstraction that uses polyhedral in this work.

The *DeepPoly* abstract domain assign to each neuron a variable that represent its abstraction. Each variable has two associate polyhedral constraints the lower and upper bounds of the variable. In addition, the variable also contains interval bounds that are derived from polyhedral constraints. Therefore, an abstraction of variable X can be written as a tuple $a = \langle a^{\leq}, a^{\geq}, l, u \rangle$ where

$$a^{\leq}, a^{\geq} \in \{x \longmapsto v + \sum w_j \cdot x_j \,|\, v \in \mathbb{R}, w_i \in \mathbb{R}\} \tag{3.1}$$

with x_j represent the abstraction of other variables and $l, u \in \mathbb{R}$ satisfying $a^{\leq} \leq x \leq a^{\geq}$ and $l \leq x \leq u$. After abstracting the given input region, *DeepPoly* propagates abstraction layer by layer according to the layer transformation until the output layer is reached. Then it checks whether the resulting abstraction always satisfy the target property. If it does, the property is verified. Otherwise, the verification fails.

To propagate abstractions, it is essential to define abstract transformers, which are used to approximate corresponding concrete layer operations. For the *Affine* operation, the abstract transformer can be defined exactly for *DeepPoly* abstract domain. In other words,

$$\forall y \in \mathit{Affine}^{\#}(a),\ \exists x \in a,\ \ \mathit{such\ \ that}\ \ y = \mathit{Affine}\,(x).$$

Now let's introduce how abstract transformer is defined for *ReLU* operation in *DeepPoly*. To balance efficiency and preciseness, *DeepPoly* checks the bounds of the neurons first and then approximate each neuron separately. In specific, assuming the current abstraction is a_j^{in}, for neuron j, a_j^{out} can be determined by the following cases:

$$\begin{cases} 0 \leq x_j^{out} \leq 0, l_j^{out} = 0, u_j^{out} = 0 & \text{if } u_j^{in} \leq 0 \\ x_j^{in} \leq x_j^{out} \leq x_j^{in}, l_j^{out} = l_j^{in}, u_j^{out} = u_j^{in} & \text{if } l_j^{in} \geq 0 \\ 0 \leq x_j^{out} \leq \frac{u_j^{in}(x_j^{in}-l_j^{in})}{(u_j^{in}-l_j^{in})}, l_j^{out} = 0, u_j^{out} = u_j^{in} & \text{if } u_j^{in} \leq -l_j^{in} \\ x_j^{in} \leq x_j^{out} \leq \frac{u_j^{in}(x_j^{in}-l_j^{in})}{(u_j^{in}-l_j^{in})}, l_j^{out} = l_j^{in}, u_j^{out} = u_j^{in} & \text{otherwise} \end{cases}$$

For the two cases that $u_j^{in} \geq 0$ and $l_j^{in} < 0$, the abstract transformer uses a convex approximation as in Fig. 3.1 where the slope of the non-vertical edge is $u_j^{in}/(u_j^{in} - l_j^{in})$ to approximate *ReLU* operation. In fact, such approximation is chosen greedily in order to minimize its area.

Next, for abstract transformers of *Sigmoid* and *Tanh* operations in *DeepPoly*, as they are similar to each other, their transformers work the same. Let $g : \mathbb{R} \to \mathbb{R}$ represents the activation function and g' is the first derivatives of g. With the current abstraction is a_j^{in}, the next abstraction a_j^{out} for neuron j can be determined as $l_j^{out} = g(l_j^{in}), u_j^{out} = g(u_j^{in})$ and

$$\begin{cases} g(l_j^{in}) \leq x_j^{out} \leq g(u_j^{in}) & \text{if } l_j^{in} = u_j^{in} \\ \begin{matrix} x_j^{out} \geq \begin{cases} g(l_j^{in}) + \lambda \cdot (x_j^{in} - l_j^{in}) & \text{if } 0 < l_j^{in} \\ g(l_j^{in}) + \lambda' \cdot (x_j^{in} - l_j^{in}) & \text{otherwise} \end{cases} \\ x_j^{out} \leq \begin{cases} g(u_j^{in}) + \lambda \cdot (x_j^{in} - u_j^{in}) & \text{if } u_j^{in} \leq 0 \\ g(u_j^{in}) + \lambda' \cdot (x_j^{in} - u_j^{in}) & \text{otherwise} \end{cases} \end{matrix} & \text{otherwise} \end{cases}$$

, with

$$\lambda = (g(u_j^{in}) - g(l_j^{in}))/(u_j^{in} - l_j^{in})$$

and

$$\lambda' = min(g'(l_j^{in}), g'(u_j^{in}))$$

The readers are referred to [30] for more details. We illustrate the *ReLU* transformer via the example in Section 2.

In addition to *DeepPoly*, there are other existing abstract inter-pretation-based approaches for neural network verification, such as *AI*^{2} [13] and *DeepZ* [28]. All these approaches share similar overall procedure, i.e., abstracting input region, then applying the abstract transformer layer by layer until the output layer, and lastly check whether resulting abstraction satisfies the target property. All these approaches would fail to verify a property if any over-approximation introduce property-violating counterexamples, as no abstraction refinement step is involved.

Chapter 4

The Approach

In this section, we present the detail of our approach.

4.1 Overall Approach

Our overall approach is shown in Algorithm 1. We maintain a verification *tree* throughout the procedure. Initially, *tree* has only one root node, which is the initial verification problem $\{\phi_I\}N_0\{\phi_O\}$. The loop from line 3 to line 17 then iteratively examines and expands the tree whenever necessary (to proceed the current verification task or decompose one verification task into multiple tasks). During each iteration, a leaf node representing an unfinished task $t = \{a^{[i]}\}N_i\{\phi_O\}$ is selected. First, it checks whether stop condition is met, i.e., timeout or too many tasks were spawn. If yes, the procedure returns a failure. Otherwise, we check whether layer i is the output layer (i.e., it does nothing other than output the result). If it is, we check whether $a^{[i]}$ satisfies ϕ_O. For local robustness property, the checking is quite straightforward. The leaf is marked finished if $a^{[i]}$ satisfies ϕ_O.

If the above case fails or layer i is not the output layer, we start working on the verification task. Considering problematic over-approximation might have been introduced in computing $a^{[i]}$, we first validate $a^{[i]}$ through procedure *validate*. Intuitively, *validate* employs optimization techniques to search for a partial counterexample 'cheaply'. The details are discussed in Section 4.2.

Algorithm 1: *DeepCause* (ϕ_I, N, ϕ_O)

```
let $a^{[0]}$ be the initial abstraction based on $\phi_I$
let *tree* be a task tree with only a root node $\{a^{[0]}\}N_0\{\phi_O\}$
**for** *each unfinished leaf* $t = \{a^{[i]}\}N_i\{\phi_O\}$ *of tree* **do**
    **if** *timeout or too many tasks are spawn* **then**
        **return** *'Failed'*
    **if** *layer* $i$ *is the output layer* **then**
        **if** $a^{[i]}$ *satisfies* $\phi_O$ **then**
            mark $t$ as finished
    counterexample $ce = validate(i, a^{[i]}, N, \phi_O)$
    **if** *ce is null* **then**
        get next abstraction $a^{[i+1]} = f_i^{\#}(a^{[i]})$
        add task $\{a^{[i+1]}\}N_{i+1}\{\phi_O\}$ as a child of $t$
    **else**
        **if** *layer* $i$ *is the input layer* **then**
            **return** *'Falsified'*
        **if** $refine(tree, i-1, N, a^{[i-1]})$ *is False* **then**
            **return** *'Failed'*
**return** *'Verified'*
```

If procedure *validate* finds no counterexamples, then no evidence shows the over-approximation introduced by $a^{[i]}$ is problematic (although there is no guarantee) and thus we proceed to abstract the next layer and construct the next verification task as the child of t. If a counterexample is at the input layer, then we know that it is a real counterexample that can invalidate the verification task. In this case, we report the task is falsified as the result. Otherwise, we start the abstraction refinement process using procedure *refine*. We leave the details on how refinement is done in Section 4.3. If the refinement fails, then we report the verification failure. Otherwise, we continue to the next unfinished task. Once all the tasks are done, the original verification task is successfully verified.

4.2 Abstraction Validation

In the following, we present details of *validate*, which decides whether an over-approximation $a^{[i]}$ of the i-th layer of N is problematic or not. Formally, $a^{[i]}$ is problematic if there exists at least one concrete input $x^{[i]} \in a^{[i]}$ such that $N_i(x^{[i]}) \vDash \neg\phi_O$.

Note that verifying whether $a^{[i]}$ is problematic equals solving the verification task itself. Thus, our goal is rather to check whether $a^{[i]}$ is problematic in a 'cheap' way. The basic idea is to sample some valuations within $a^{[i]}$ and check whether any of them are counterexamples. Due to the geometrically complicated region defined by $a^{[i]}$, trivial sampling is usually ineffective. In the following, we show how we could sample and search for counterexamples in a systematic, effective, and efficient way.

First, let's consider the question how to sample valid valuations of $x^{[i]}$ within *DeepPoly* $a^{[i]}$. We can easily obtain all possible sample of $x^{[i]}$ by using the lower bound and upper bound in $a^{[i]}$. The sample is guaranteed to be valid and over-approximate all the possible value of $x^{[i]}$.

Once obtaining valid samples, our approach attempts to search for counterexamples through pushing these samples towards property-violating zone. Through drawing inspiration from the recently developed effective method for adversarial perturbation [24], we formulate abstraction validation as an optimisation problem and leverage the gradient information to efficiently generate counterexamples. In particular, the idea is to devise a loss function to measure how far the current sample is to violate the target property, and drive the current sample toward property-violating zone through decreasing the value of loss function. In fact, we adopt the projected gradient descent (PGD) [23], a standard method for large-scale constrained optimization, to optimize the loss function based on gradient information. Meanwhile, the gradient information can be efficiently calculated through the chain rule. Formally, we have

$$\frac{\partial Loss(N_i \circ x_i, \phi_O)}{\partial x_i} = \frac{\partial Loss(N_i \circ x_i, \epsilon, \phi_O)}{\partial (N_i \circ x_i)} \times \frac{\partial (N_i \circ x_i)}{\partial x_i}$$

where the two ingredients on the right side can be collected or calculated efficiently.

Algorithm 2 shows the details of our abstraction validation. The loop from line 2 to line 10 allows us to search from multiple random seeds until a timeout occurs. At line 3, we generate a random seed x_i^* in range $[l^{[i]}, u^{[i]}]$. The loop from line 4 to line 10 then navigates through the space of x_i^* iteratively. In particular, after computing the input x_i^*,

Algorithm 2: *validate*$(i, a^{[i]}, N, \phi_O)$

```
1  let counterexample $ce$ be $null$
2  while $not\ time\ out$ do
3      generate random seed $x_i^*$ within range $[l^{[i]}, u^{[i]}]$
4      while $less\ than\ M\ iterations$ do
5          if $N_i(x_i^*)$ $violates$ $\phi_O$ then
6              $ce = x_i^*$
7              return $ce$
8          compute gradient $grad = \frac{\partial Loss(N_i \circ x_i, \phi_O)}{\partial x_i}$
9          $x_i^* = x_i^* - \alpha\ sgn(grad)$
10         clip $x_i^*$ to fit in range $[l^{[i]}, u^{[i]}]$
11 return $ce$   /* null in this case */
```

we check whether x_i^* could lead to property violation, i.e., $N_i(x^*)$ violates ϕ_O at line 5. If it is, we return x_i^* as a counterexample. Otherwise, we follow the gradient to search through the space. That is, at line 9, we modify x_i^* according to a loss function which is discussed as follows.

Lastly, we briefly discuss how loss function is constructed. For the robustness analysis, where our goal is to generate samples with distinct prediction (i.e., label) other than the original label t, a simple loss function can be defined as below:

$$Loss(f, x, \phi_O) \quad = \quad O_t - \max_{j \neq t}(O_j) \quad where \quad O = f(x).$$

Apparently, decreasing the value of the loss function navigate the input x towards the property-violating zone. Once a negative loss value is reached, the optimization procedure return x as a counterexample (a.k.a. adversarial examples in the neural network attack scenario).

Except the robustness property, ϕ_O can be given in a more complicated form such as a constraint in SMTLIB syntax, wherein defining similar loss functions is also possible. In fact, there are systematic ways to transform any quantifier-free constraint into a loss function in linear time. As this is not central to our discussion and thus we refer interested readers to [12] for details.

Algorithm 3: $refine(tree, i, N, a^{[i]})$

```
1 $measure = causal(i, N_i)$
2 get neurons $js = k\_select(measure, a^{[i]}, k = 1)$
3 if $js$ is empty then
4     return False
5 get refined abstractions $as = k\_split(a^{[i]}, js)$
6 if $as$ is empty then
7     return False
8 replace sub-tree rooted by $a^{[i]}$ with $as$
9 return True
```

4.3 Abstraction Refinement

Now, we present details on how abstraction refinement is done in our framework, i.e., the details of procedure *refine*, which is shown in Algorithm 3. The inputs are the layer i, the network N, the current abstraction represented in the form of a tree $tree$, and the abstraction $a^{[i]}$ at layer i.

Suppose the abstraction $a^{[i+1]}$ is invalid with the witness of counterexample ce. Apparently partitioning $a^{[i+1]}$ itself can not make it more precise. Our idea is to partition the very abstraction before $a^{[i+1]}$. In specific, we refine $a^{[i+1]}$ by partitioning k neurons of abstraction $a^{[i]}$ (notice the index at line 16 of Algorithm 1). In the implementation, we set $k = 1$. The k neurons can be regarded as the most effective neurons on the neural network output. To select these neurons, our work mainly consider their causal effect (Section 4.4). Intuitively, the causal effect can be regarded as an indicator on how sensitive the output of neural network is with regards to the neuron. Thus, refining the neuron with maximum causal effect is more likely to change the output so that ce is no longer a spurious counterexample (i.e., its output according to the abstraction would satisfy ϕ_O).

Next, we describe how neurons are selected and partitioned for different layer operations. Note that we only consider the activation operation where the abstraction is not exact so *Affine* layer will not be chosen to partition.

Neuron selection and partition for *ReLU* operation. Given a $ReLU$ function $x_j^{[i+1]} = ReLU\,(x_j^{[i]})$, approximation is only introduced for the neuron j such that $l_j^{[i]} < 0$

and $u_j^{[i]} > 0$. Therefore, in function *k_select* we only consider these neurons. Once we picked k neurons, we partition each of them at $x_j^{[i]} = 0$ so that the resulting abstraction for those neurons are exact.

Neuron selection and partition for *Sigmoid* and *Tanh* operation. Different from *ReLU* activation function, *Sigmoid* and *Tanh* are only exact when $l_j^{[i]} = u_j^{[i]}$ so for the most case, all neurons can be considered. In this work, we partition each neuron at the value $(l_j^{[i]} + u_j^{[i]})/2$. We notice that unlike *ReLU*, the next approximation on the refined neuron may not become exact after the refinement. As a result, to prevent the algorithm to partition the neuron infinite, each neuron is only partitioned once. We admit that there may be a more systematic way to choose the partition values for *Sigmoid* and *Tanh* functions and left it to the future work.

4.4 A Causality-based Selection

In this section, we will explain how we rank each neuron for refinement. Since the number of refining tasks can grow exponentially with the size of the network, it is important that we need to find the neurons that have the most impact on the output to refine. *DeepCause* thus adopts a causality-based method to analyse all neurons in the considering hidden layer.

In recent years, causality analyse has gain increasing attention in neural network interpretation [7, 14, 43, 26, 33]. Many different approaches have used casual attribution to explain the relationship between the components in neural networks. Compare to traditional method like gradient, which suffer from sensitivity and induce causal effects biased by other input features [7], the causal approach can identify the cause and effect between network's components and thus be able to measure the impact of hidden neurons on the output of the network.

In the following, we go over some concepts which is necessary to understand causality analysis in this work.

Definition 4.1. (Structural Causal Models) [7]. A Structural Causal Model (SCM) is a 4-tuple (X, U, f, P_u) where, X is a finite set of endogenous variables, usually the observable random variables in the system; U is a finite set of exogenous variables, usually treated

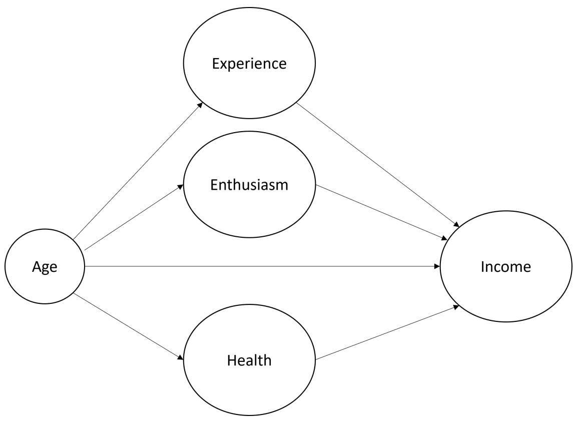

Figure 4.1: An example casual graph.

as unobserved or noise variables; f is a set of functions $[f_1; f_2; ..., f_n]$, where n refers to the cardinality of the set X and these functions define causal mechanisms, such that $\forall x_i \in X; x_i = f_i(Par, u_i)$. The set Par is a subset of $X - \{x_i\}$ and $u_i \in U$. Finally, P_u defines a probability distribution over U.

The example in Figure 4.1 is a casual graph that shows a study about income of people, with nodes being variables and edges represent cause-effect relationship. In this graph, age is an exogenous variable while experience, enthusiasm and heath that come from age are a set of endogenous variable. The outcome of this is income. As can be seen from the graph, age factor has direct effect on experience (more age usually come with more experience), enthusiasm (young people have more willing and drive to work) and heath (heath often weaken as time go on). All of these elements can affect what job you have and in turn, your income. Furthermore, your own age can have direct impact on your income.

Based on the result in [7], we know that a neural network can be represented in form of a SCM by considering a set of exogenous random variables that act as causal factors for input neurons, then considering the neurons in the previous layer as causal factors for the neurons in the next layer. We then define the causal attribution of each neuron in the network as follows.

Definition 4.2 (Causal Attribution)**.** We denote the measure of the causal attribution of given neural network N as y. The Causal Attribution of a neuron x in $N's$ to y is:

$$ACE^{y}_{do(x)} = \mathbb{E}_x[\mathbb{E}_y[y|do(x = \beta)]] \tag{4.1}$$

Algorithm 4: *causal*(i, N_i)

```
1 let $t$ be the label of the original input
2 generate random seed $x_i^*$ within range $[l^{[i]}, u^{[i]}]$
3 $u^{[k]} = N_i(x_i^*)$
4 for neuron $j$ in $N_i$ do
5     $\beta = uniform(l_j^{[i]}, u_j^{[i]})$
6     $v_j^{[k]} = N_i(x_i^*, do(x_i^j = \beta))$
7 $y = |u_t^{[k]} - v_t^{[k]}|$
8 return $y$
```

We notice that our formula is different from [7] because in [7], the authors want to compute the causal attribution according to a fixed value β of the neuron, while in our work, we want to compute the average attribution based on the neuron's interval. In the implementation shown in Algorithm 4, to compute the right-hand side, we simply apply uniform sampling to collect a set of values of the neuron based on its interval (line 5) (which is computed based on the abstract interpretation as presented in Section 3.3). Then for each value β, we apply the *do* operation (i.e., set the value of the variable x representing the considering neuron to β while keeping the values of other neurons in the same layer unchanged). Let $u^{[k]}$ in line 3 be the output vector according to the random input (from the layer with the *do* operation) and $v^{[k]}$ in line 6 be the output vector according to the input after applying the *do* operation, we define $y = \left|u_t^{[k]} - v_t^{[k]}\right|$ with t is the label of the original input in line 7. Intuitively, the variable y indicates how much the value of the output at t changes according to the *do* operation.

Example 4.3. Now we demonstrate how Algorithm 4 work by using $x^{[5]}$ in example 2.3. In example 2.3, after determine that $x^{[5]}$ need to refine, we choose $x_0^{[5]}$ without using casual analysis because $x_1^{[5]}$ is precise. However, in this example, we disregard that and calculate the casual value of each features regardless.
Let generate random seed in range of $x^{[5]}$ where $l_0^{[5]} = 0, u_0^{[5]} = 0; l_1^{[5]} = 1, l_1^{[5]} = 5$:

$$x_5^* = \begin{bmatrix} 0.5 & 4 \\ 1.5 & 2 \\ 1.75 & 3.5 \end{bmatrix}$$

As such, we have the output value of :

$$u = \begin{bmatrix} 0.5 & 4 \\ 1.5 & 2 \\ 1.75 & 3.5 \end{bmatrix} * \begin{bmatrix} -2 & 2 \\ 1 & -2 \end{bmatrix} = \begin{bmatrix} 3 & -7 \\ -1 & -1 \\ 0 & -3.5 \end{bmatrix}$$

Next, we apply uniform sampling on $x_0^{[5]}$: $\beta_0 = [0, 1, 2]$

$$do(x_0^{[5]} = \beta) = \left[\begin{bmatrix} 0 & 4 \\ 0 & 2 \\ 0 & 3.5 \end{bmatrix}, \begin{bmatrix} 1 & 4 \\ 1 & 2 \\ 1 & 3.5 \end{bmatrix}, \begin{bmatrix} 2 & 4 \\ 2 & 2 \\ 2 & 3.5 \end{bmatrix} \right]$$

$$v_0 = N_5(x_5^*, do(x_0^{[5]} = \beta)) = \left[\begin{bmatrix} 4 & -8 \\ 2 & -4 \\ 3.5 & -7 \end{bmatrix}, \begin{bmatrix} 2 & -6 \\ 0 & -2 \\ 1.5 & -5 \end{bmatrix}, \begin{bmatrix} 0 & -4 \\ -2 & 0 \\ -0.5 & -3 \end{bmatrix} \right]$$

The label of the original input is 0. Therefore, we have the result y as following:

$$u[t] - v_0[0] = \left[\begin{bmatrix} -1 \\ -3 \\ -3.5 \end{bmatrix}, \begin{bmatrix} 1 \\ -1 \\ -1.5 \end{bmatrix}, \begin{bmatrix} 3 \\ 1 \\ 0.5 \end{bmatrix} \right]$$

$$y_0 = |u[0] - v_0[0]| = \left(\left| \frac{-1-3-3.5}{3} \right| \left| + \frac{1-1-1.5}{3} \right| + \left| \frac{3+1+0.5}{3} \right| \right) /3 = 1.5$$

We apply the same process to $x_1^{[5]}$ with $\beta_1 = [1, 3, 5]$. The result is $y_1 = 1.389$. According to the 2 values, we can conclude that feature $x_0^{[5]}$ has more impact to the label 0 compare to feature $x_1^{[5]}$. As a result, $x_0^{[5]}$ is chosen to refine.

4.5 Soundness

Before closing this section, we briefly analyze our approach in terms of soundness. In our approach, the primary means of verifying neural network is abstraction, and we thus focus on how abstractions are manipulated during the verification. In fact, the abstractions in our approach can only be manipulated in the following three ways: (a) created from a given concrete input region; (b) computed through applying abstract transformer on existing abstractions and (c) refined by partitioning existing abstractions. It is not hard to guarantee the abstraction from (a) and (b) safely over-approximate corresponding concrete states. For case (c), suppose we split an abstraction $a^{[i]}$ and get a set of abstraction as. It is easy to prove that $\forall x^{[i]} \in a^{[i]}$, there exists at least one abstraction $a^{[i],pr} \in as$ that $x^{[i]} \in a^{[i],pr}$. Therefore, the soundness is proved[1].

[1]As our implementation is based on ELINA library which is sound with respect to floating point semantics, our implementation gets this benefit and is thus sound with respect to floating point semantics.

Chapter 5

Evaluation

As a proof-of-concept demonstration, we have implemented our approach as a prototype tool named *DeepCause* on top of *DeepPoly* abstract domain. The main procedure of *DeepCause* is written in Python with more than 7500 line of code while the underlying *DeepPoly* computation is performed by the ELINA library [31, 32].

To evaluate our approach, in this section, we would like to answer the following three research questions (RQs) :

RQ1: How does *DeepCause* perform compare to *DeepPoly*?

RQ2: How does *DeepCause* perform when using gradient-selection?

RQ3: How efficient is *DeepCause* compare to other approaches?

5.1 Experimental Setup

To answer above research questions, we have conducted an extensive set of experiments as follows.

Benchmarks. We collected a benchmark suite of verification tasks across neural network models trained on MNIST [20] image dataset. MNIST consists of 60000 grayscale images of handwritten digits, whose resolution is 28×28 pixels. The images show white digits on a black background. During the training, we vary the model size in order to test the scalability of our approach. In specific, the number of hidden layers in the models varies from 3 to 5, and the number of neurons for each hidden layer varies from 10 to 50 (step size if 10). The activation functions in the hidden layers of the networks are either *ReLU*, *Sigmoid*, or *Tanh*. In total, we have 45 networks. The first 100 images from the test set are chosen as the test inputs. We notice that for each network model, only the inputs which are classified correctly by the network are used for robustness evaluation. In total, there are 4380 verification tasks.

In addition, we apply our approach *DeepPoly* and *RefinePoly* to compare our performance. *RefinePoly* is a network robustness verified approach that combines the strength of MILP solver and LP relaxation with fast overapproximation methods (which is *DeepPoly* in this case). This method uses the abstract domain as input for MILP solver to refine the boundary of the domain which in turn make the abstraction tighter which result in better performance. Therefore, *RefinePoly* is a refinement-based approach that improve *DeepPoly* precision at the cost of it's runtime. We implement our method into *RefinePoly* by splitting the neuron chosen by heuristic and use MILP solver and LP relaxation to further refine the split boundary. We apply our approach on *RefinePoly* to see how effective it is with other method.

Robustness properties. We consider the robustness properties using L_∞-norm [6] distance. This mean that the adversarial region contains all perturbed images x' where each pixel x'_i has a distance of at most ϵ from the corresponding pixel x_i in the original input x. For each test input, we try to find the maximum verified robustness bound according to each verifying approach. The step size of the bound is 0.001.

Experimental Configuration All of our experiments were performed on a Ubuntu 20.04 machine with a 6-core CPU and 16GB memory. Considering the efficiency and termination issues, we set 128 as the maximum number of tasks for each layer in the network. Moreover, each neuron in a layer is only refined once. Thus, a verification problem can be either verified, falsified, or exceeding the maximum refinement limit.

5.2 Experiment Results

		$n = 10$		$n = 20$		$n = 30$		$n = 40$		$n = 50$	
		$\#_c$	$\%_c$	$\#_c$	$\%_c$	$\#_c$	$\%_c$	$\#_c$	$\%_c$	$\#_c$	$\%_c$
ReLU	$k = 3$	63	18.3	65	12	70	12.3	77	9.4	59	8.5
	$k = 4$	66	15.4	74	13.4	69	14.5	68	10.6	60	8.3
	$k = 5$	58	13.6	70	13.4	78	12.7	68	10.3	57	9.2
Sigmoid	$k = 3$	66	10.8	49	9	48	7.8	41	7.7	25	7.3
	$k = 4$	51	9.9	45	8.1	35	8.2	32	8.8	28	7.9
	$k = 5$	68	12.4	54	8.8	32	7	29	8.7	27	6.6
Tanh	$k = 3$	74	15.6	55	11.7	28	13.8	25	12.6	21	14.1
	$k = 4$	78	15.7	37	10.4	25	14.6	15	15.3	15	16.3
	$k = 5$	66	15.08	28	16.8	10	16	8	17.1	5	24.9

Table 5.1: Comparison between *DeepPoly* and *DeepCause*

To answer **RQ1**, we compare the performance of *DeepCause* with *DeepPoly* on the benchmark networks. We report the results based on two measures:

$\#_c$ the number of the inputs whose verified robustness bounds get improved by *DeepCause*;

$\%_c$ the average improved percentage of the robustness bounds on the above inputs.

For example, if there is one improved input and the verified robustness are 0.05 and 0.06 by *DeepPoly* and *DeepCause* respectively, then the corresponding results are $\#_c = 1$ and $\%_c = (0.06 - 0.05)/0.05 = 20\%$. We notice that as the underlying engine of *DeepCause* is based on *DeepPoly*, the worst case is that *DeepCause* has the same effectiveness as *DeepPoly*.

The results are shown in Table 5.1, which the first column shows the activation function, the second column shows the number of hidden layers, and the remaining columns show the results according to the number of neurons in each hidden layer. From the results, we see that *DeepCause* can improve the verified robustness bounds of *DeepPoly* for all networks. The number of improved testcases is at least 5 to as large as 77 while the average improved bounds can range from 6.6% at worst to 24.9% at best. Totally, there

		$n = 10$				$n = 20$				$n = 30$				$n = 40$				$n = 50$			
		$\#_g$	$\#_c$	$\%_g$	$\%_c$	$\#_g$	$\#_c$	$\%_g$	$\%_c$	$\#_g$	$\#_c$	$\%_g$	$\%_c$	$\#_g$	$\#_c$	$\%_g$	$\%_c$	$\#_g$	$\#_c$	$\%_g$	$\%_c$
ReLU	$k = 3$	10	10	4.85	8.88	18	18	5.76	6.71	11	28	6.19	6.49	6	41	5.58	5.85	17	28	5.57	6.72
	$k = 4$	11	6	5.51	6.03	14	14	6.78	5.75	14	25	6.91	5.94	9	33	6.69	6.57	11	23	6.76	5.83
	$k = 5$	5	13	5.31	7.86	11	17	5.78	7.01	9	31	8.48	7.01	5	22	6.47	7.4	7	29	7.16	7.17
Sigmoid	$k = 3$	16	13	7.61	9.46	1	28	7.14	8.13	0	28	_	7.59	0	23	_	6.95	1	13	7.69	6.85
	$k = 4$	11	14	6.98	4.89	1	26	8.33	7.72	0	22	_	7.65	0	18	_	8.36	3	20	9.39	7.38
	$k = 5$	0	49	_	10.02	3	37	10.08	8.91	4	22	4.6	7.26	1	14	4.55	6.97	1	12	5.26	6.39
Tanh	$k = 3$	2	51	7.29	12.63	0	43	_	11.43	0	21	_	13.69	0	21	_	13.33	0	21	_	14.06
	$k = 4$	0	54	_	12.81	0	29	_	9.99	0	20	_	13.8	0	14	_	15.72	0	14	_	16.46
	$k = 5$	0	44	_	14.34	0	23	_	16.38	0	10	_	16.03	0	5	_	20.02	0	5	_	24.86

Table 5.2: Comparison between *DeepGrad* and *DeepCause*

are 2122 improved tasks (i.e., 48%). The results also show that *DeepCause* works best for the *ReLU* networks where it improves the bounds of 1002 tasks, nearly the half of all improved tasks. This is expected because the one-time refinement (as in our implementation) for each hidden neuron in the *ReLU* networks yields the exact approximation in the next step, while the one-time refinement in *Sigmoid* and *Tanh* networks does not have this property. We also see that the effectiveness of *DeepCause* drops on large networks. The number of tasks improved falls from around 50-80 task in $n = 10$ to around 5-25 in $n = 50$ for the *Sigmoid*, *Tanh* networks and the average percentage also decreases from around 15% to around 8% for *ReLU* and *Sigmoid* networks. Again, the results are expected due to the fact that with large networks, there may be more neurons which need to be refined at the same time, which will generate more tasks at the considering layer, and with a limited number of tasks for a layer, we may not refine all the necessary neurons (especially for the *Sigmoid* and *Tanh* networks, where the refinement does not yield the exact approximation for the neurons).

To answer **RQ2**, we substitute the function *k_select* in Algorithm 3 by another heuristic, in which we choose the neurons to refine based on the gradient. That is, with the counterexample *ce*, we compute the gradient of each neuron in the considering layer according to the output value of target label, then choose the neurons based on their absolute gradient from highest to lowest. We call the implementation with this heuristic *DeepGrad* and compare its performance with *DeepCause*. Unlike RQ1, there may be test inputs which *DeepGrad* has better performance than *DeepCause*, so we use the following measures:

$\#_g$, $\#_c$ the number of the inputs whose verified robustness bound get improved by *DeepGrad* compare to *DeepCause* and vice versa;

$\%_g$, $\%_c$ the average improved percentage of the robustness bound by *DeepGrad* compare to *DeepCause* and vice versa.

The results are shown in Table 5.2. We notice the $\%_g$ column is indefinable in case $\#_g = 0$. From the table, we can see that *DeepCause* and *DeepGrad* are complement to each other. However, it is clear that *DeepCause* has better results, especially when the the sizes of the networks increase and for *Sigmoid* and *Tanh* networks.*DeepCause* has better testcases for every network while *DeepGrad* doesn't have any in 19 networks, all of which is either *Sigmoid* or *Tanh* networks. For *ReLU* networks, the different in the number of improved input between *DeepCause* and *DeepGrad* is around 0-6 in n=10, 20 and it grows to around 15-30 in n=40, 50. In summary, *DeepGrad* has better results in 202 tasks (i.e., 5%) and *DeepCause* has better results in 1052 tasks (i.e., 24%). Moreover, the average improved percentage of *DeepCause* is slightly better than *DeepGrad* in more networks in the remaining definable cases. This result shows that the causality approach not only work better than gradient approach in general but it is also a more suitable approach for *Sigmoid*, *Tanh* and big network.

In addition of *DeepPoly*, we also apply our method and heuristic to *RefinePoly*. Due to limit resource, we are only able to test it on *ReLU* network but the result shows in Table 5.4 indicates a similar pattern. *DeepCause* has better performance on both categories in every network except one (network ReLU_4_10) and the different become larger with larger network. Table 5.4 shows that *DeepCause* verifies better robustness in 432 tasks (i.e., 79%) with maximum percentage is 8.89% compare to *DeepGrad* with 114 tasks (i.e., 21%) and 6.41% respectively. In conclusion, both result on *DeepPoly* and *RefinePoly* shows that our method can be appiled and further enhance the performance of existing methods.

Finally, to answer **RQ3**, we report the total running time of *DeepPoly*, *DeepGrad*, and *DeepCause* to verify the max robustness bound according to each tool. From the table, we can see that *DeepCause* spends more time than *DeepGrad* and *DeepPoly*, which is expected due to *DeepCause* has a complex refinement strategy compare to other approaches. The running time of *DeepCause* in general is about 2-5 times higher than *DeepGrad* with the longerst time is 27626s and 17333s, respectively. Consider that *DeepCause* has much better effectiveness, i.e., verify larger robustness bounds, we believe the more running time is reasonable. The time of *DeepCause* usually become larger with the size of the network except when the network become to big or with *Sigmoid*, *Tanh* network where

		$n=10$			$n=20$			$n=30$			$n=40$			$n=50$		
		t_p	t_g	t_c	t_p	t_g	t_c	t_p	t_g	t_c	t_p	t_g	t_c	t_p	t_g	t_c
ReLU	$k=3$	9.5	421.8	615.1	16.2	1859	2123	19.2	3897	6806	24.0	2362	8755	27.9	1847	9739
	$k=4$	11.7	748.5	1024	22.6	10472	8927	32.9	17333	27626	37.2	2611	15424	40.7	4390	8805
	$k=5$	14.0	551.8	1356	31.6	12729	22210	39.5	6900	16811	58.5	7166	18429	53.6	1815	6636
Sigmoid	$k=3$	10.7	2067	4756	21.5	872.6	4401	26.2	1389	8321	31.3	3138	7074	35.7	820.2	5294
	$k=4$	11.2	3435	6493	30.3	1774	3665	39.7	1411	6290	44.8	2096	6229	62.2	4166	6420
	$k=5$	16.7	2874	9272	37.2	3649	1871	53.4	988.7	7481.2	70.7	1139	3096	79.1	6478	3550
Tanh	$k=3$	8.8	2625	9796	16.3	1003	8086	21.6	1216	7499	25.8	93.3	9104	31.2	31.2	11034
	$k=4$	10.7	6421	15265	24.5	458.5	6016	33.1	301.5	9213	40.6	64.6	6408	49.8	174.7	9152
	$k=5$	13.7	2891.4	12606	34.3	293.1	9797	45.8	45.8	2619	59.2	213.1	1250	71.7	71.7	1565

Table 5.3: The running time (second) of *DeepPoly*, *DeepGrad*, and *DeepCause* to verify the max robustness bound

		$n=10$				$n=20$				$n=30$				$n=40$				$n=50$			
		$\#_g$	$\#_c$	$\%_g$	$\%_c$	$\#_g$	$\#_c$	$\%_g$	$\%_c$	$\#_g$	$\#_c$	$\%_g$	$\%_c$	$\#_g$	$\#_c$	$\%_g$	$\%_c$	$\#_g$	$\#_c$	$\%_g$	$\%_c$
ReLU	$k=3$	8	16	5.26	7.29	11	22	4.9	6.8	9	27	5.17	7.66	5	25	4.43	6.39	1	19	3.7	5.58
	$k=4$	14	1	4.88	2.78	15	28	5.51	7.2	12	32	6.41	6.43	6	39	5.05	6.59	5	38	4.05	5.96
	$k=5$	8	9	4.57	4.97	13	43	5.37	8.89	2	47	3.62	7.08	2	41	3.79	6.76	3	45	5.06	6.8

Table 5.4: Comparison between *RefinePoly_DeepGrad* and *RefinePoly_DeepCause*

our approach become ineffective which make the time go down. We can clearly see the trend of this in the Table 5.3 where the running time of *DeepCause* and *DeepGrad* increase from column n=10 to n=30 and then decrease to column n=50. We also notice that when we verify the max robustness bound of *DeepPoly* with *DeepGrad* and *DeepCause*, the running time of three approaches should be similar. It is shown at the *Tanh* networks with $n = 50$, $k = 3$ or $k = 5$, the running time of *DeepGrad* and *DeepPoly* are the same because *DeepGrad* does not have any improved robustness bound in these networks compare to *DeepPoly*.

Chapter 6

Related Work

Formal local robustness verification for neural networks. To be able to prove the robustness of neural networks, the researchers employ many different certification techniques. These already existing verifiers can be broadly classified as either complete or incomplete. Complete verifiers are exact, i.e., if the verifier fails to certify a network then the network is non-robust (and vice-versa). Therefore, these verifiers do not have false positives but they have limited scalability and cannot handle neural networks containing more than a few hundred hidden neurons. This is because complete verifiers are based on on computationally expensive methods like SMT solving [11, 38], mixed integer linear programming [35] or input refinement [39]. On the other hand, incomplete verifiers can produce false positives which mean that they are sound but may fail to prove robustness even if it holds. However, they scale much better than complete verifiers due to a variety of over-approximation method that they employed like duality [10], linear approximation [40, 41], and abstract interpretation [13, 28]. In summary, there are 2 key challenges that need to face when creating a new verifier: *Scalability* and *Precision.* A sound analysis of neural networks need to be able to scale to large network while maintaining its precision. Therefore, in this thesis, we aim for the sweet spot between precision and scalability.

Neural Network Verification based on Abstraction. This work is closely related to the many recent proposals on neural network verification. To list a few, AI^2 is the very first approach to employ Zonotope abstraction to verify neural networks and shows it's promise compare to other approaches, and then *DeepZ* improves its precision via better abstract transformers for activation functions. Later, *DeepPoly* devises a new abstract domain based on polyhedra and interval suitable for neural network verification and proves that it is faster and more effective than others state-of-the-art method. Recently, researchers propose the Star-based approach [37, 36] to approximate neural network behavior, and claims to achieve the state-of-the-art performance. Different from our approach, all these approaches verify neural network without refinement.

The idea of splitting the bounds of inputs or hidden neurons has been explored in previous works [17, 11, 39, 38, 22] and can be fitted into a unified Branch-and-Bound framework [5]. Therein, [17] encodes the verification problem as a constraint and precisely solving the constraint, which is very different from abstraction-based methods while [39] only partition the input region with the guidance of heuristics. In addition, while both of the researches [11, 38] branch the problem by adding linear constraint in the solver at *ReLU* layer, [38] chooses the neuron to split by gradient heuristic, whereas [11] doesn't employ specialized heuristics for it. For other works, *DeepCause* is different from them as we leverage counterexamples to perform the refinement as soon as possible. Our approach attempts to find out the root cause of the verification failure and operates on the internal abstractions to fix it. From this perspective, our approach can also be extended to help debug the abstraction-based verification. Moreover, we use causality reasoning to choose the neurons to refine, which shows to be more effective than gradient.

Program Verification CEGAR is a technique widely applied in program verification scenarios [8, 16, 9, 2, 3, 15, 21]. The general procedure is to abstract program states first and applying refinement to get rid of spurious counterexamples. Like in this work, the important questions include when and how to apply the refinement.

Chapter 7

Conclusion and Future Work

In this work, we propose *DeepCause*, an approach to verify neural networks through abstraction refinement, which naturally inherits the idea of *CEGAR*. In particular, we develop an optimisation-based searching procedure to effectively identify problematic over-approximation and a causality-based heuristics to help select most likely problematic components of an abstraction for refinement. According to the evaluations, *DeepCause* could improve state-of-the-art methods based on the same abstract domain.

In terms of the future works, we would like to explore two directions. On the one hand, we plan to extend our approach to support more complicated abstract domains, such as *kReLu* [27], so as to further advance the robustness bound in neural network verification. On the other hand, as the evaluations shows *DeepCause* has limitation in the scalability, we would like to improve our approach to handle neural networks in large scales and thus it can work on neural network in more realistic scenarios.

Bibliography

[1] Thomas Ball, Vladimir Levin, and Sriram K Rajamani. A decade of software model checking with slam. *Communications of the ACM*, 54(7):68–76, 2011.

[2] Thomas Ball and Sriram K Rajamani. Automatically validating temporal safety properties of interfaces. In *International SPIN Workshop on Model Checking of Software*, pages 102–122. Springer, 2001.

[3] Dirk Beyer, Thomas A Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker b last. *International Journal on Software Tools for Technology Transfer*, 9(5-6):505–525, 2007.

[4] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, et al. End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316*, 2016.

[5] Rudy R Bunel, Ilker Turkaslan, Philip Torr, Pushmeet Kohli, and Pawan K Mudigonda. A unified view of piecewise linear neural network verification. In *Advances in Neural Information Processing Systems*, pages 4790–4799, 2018.

[6] Nicholas Carlini and David Wagner. Towards evaluating the robustness of neural networks. In *2017 ieee symposium on security and privacy (sp)*, pages 39–57. IEEE, 2017.

[7] Aditya Chattopadhyay, Piyushi Manupriya, Anirban Sarkar, and Vineeth N. Balasubramanian. Neural network attributions: A causal perspective. In *Proceedings of the 36th International Conference on Machine Learning*, volume 97, pages 981–990. ICML 2019, Long Beach, California, USA, 2019.

[8] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *International Conference on Computer Aided Verification*, pages 154–169. Springer, 2000.

[9] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM (JACM)*, 50(5):752–794, 2003.

[10] Krishnamurthy Dvijotham, Robert Stanforth, Sven Gowal, Timothy Mann, and Pushmeet Kohli. A dual approach to scalable verification of deep networks. In *In Proc. Uncertainty in Artificial Intelligence (UAI)*, pages 162–171, 2018.

[11] Ruediger Ehlers. Formal verification of piece-wise linear feed-forward neural networks. In *International Symposium on Automated Technology for Verification and Analysis*, pages 269–286. Springer, 2017.

[12] Zhoulai Fu and Zhendong Su. Xsat: a fast floating-point satisfiability solver. In *International Conference on Computer Aided Verification*, pages 187–209. Springer, 2016.

[13] 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.

[14] Michael Harradon, Jeff Druce, , and Brian E. Ruttenberg. Causal learning and explanation of deep neural networks via autoencoded activations. *abs/1802.00541*, 2018.

[15] Thomas A Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth L McMillan. Abstractions from proofs. *ACM SIGPLAN Notices*, 39(1):232–244, 2004.

[16] Thomas A Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 58–70, 2002.

[17] 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.

[18] P.-J. Kindermans, S. Hooker, M. Alber J. Adebayo, K. T. Schutt, S. Dahne, D. Erhan, and B. Kim. The (un) reliability of saliency methods. *arXiv preprint arXiv:1711.00867*, 2017.

[19] Konstantina Kourou, Themis P Exarchos, Konstantinos P Exarchos, Michalis V Karamouzis, and Dimitrios I Fotiadis. Machine learning applications in cancer prognosis and prediction. *Computational and structural biotechnology journal*, 13:8–17, 2015.

[20] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[21] Jiaying Li, Jun Sun, Li Li, Quang Loc Le, and Shang-Wei Lin. Automatic loop-invariant generation and refinement through selective sampling. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pages 782–792. IEEE Press, 2017.

[22] Jingyue Lu and M Pawan Kumar. Neural network branching for neural network verification. *arXiv preprint arXiv:1912.01329*, 2019.

[23] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards deep learning models resistant to adversarial attacks. *arXiv preprint arXiv:1706.06083*, 2017.

[24] MadryLab. Mnist adversarial examples challenge. `https://github.com/MadryLab/mnist_challenge`, 2017. [Online; accessed 01-July-2020].

[25] Kenneth L McMillan. Lazy annotation for program testing and verification. In *International Conference on Computer Aided Verification*, pages 104–118. Springer, 2010.

[26] Tanmayee Narendra, Anush Sankaran, Deepak Vijaykeerthy, and Senthil Mani. Explaining deep learning models using causal inference. *abs/1811.04376*, 2018.

[27] Gagandeep Singh, Rupanshu Ganvir, Markus Püschel, and Martin Vechev. Beyond the single neuron convex barrier for neural network certification. In *Advances in Neural Information Processing Systems*, pages 15098–15109, 2019.

[28] Gagandeep Singh, Timon Gehr, Matthew Mirman, Markus Püschel, and Martin Vechev. Fast and effective robustness certification. In *Advances in Neural Information Processing Systems*, pages 10802–10813, 2018.

[29] Gagandeep Singh, Timon Gehr, Markus Püschel, and Martin Vechev. Boosting robustness certification of neural networks. In *International Conference on Learning Representations*, 2018.

[30] Gagandeep Singh, Timon Gehr, Markus Püschel, and Martin Vechev. An abstract domain for certifying neural networks. *Proceedings of the ACM on Programming Languages*, 3(POPL):41, 2019.

[31] Gagandeep Singh, Markus Püschel, and Martin Vechev. Making numerical program analysis fast. *ACM SIGPLAN Notices*, 50(6):303–313, 2015.

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

[33] Bing Sun, Jun Sun, Hong Long Pham, and Jie Shi. Causality-based neural network repair. 2022.

[34] M. Sundararajan, A. Taly, and Q Yan. Axiomatic attribution for deep networks. *arXiv preprint arXiv:1703.01365*, 2017.

[35] Vincent Tjeng, Kai Xiao, and Russ Tedrake. Evaluating robustness of neural networks with mixed integer programming. *arXiv preprint arXiv:1711.07356*, 2017.

[36] Hoang-Dung Tran, Stanley Bak, Weiming Xiang, and Taylor T Johnson. Verification of deep convolutional neural networks using imagestars. *arXiv preprint arXiv:2004.05511*, 2020.

[37] Hoang-Dung Tran, Diago Manzanas Lopez, Patrick Musau, Xiaodong Yang, Luan Viet Nguyen, Weiming Xiang, and Taylor T Johnson. Star-based reachability analysis of deep neural networks. In *International Symposium on Formal Methods*, pages 670–686. Springer, 2019.

[38] Shiqi Wang, Kexin Pei, Justin Whitehouse, Junfeng Yang, and Suman Jana. Efficient formal safety analysis of neural networks. In *Advances in Neural Information Processing Systems*, pages 6367–6377, 2018.

[39] 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.

[40] Tsui-Wei Weng, Huan Zhang, Hongge Chen, Zhao Song, Cho-Jui Hsieh, Duane Boning, Inderjit S Dhillon, and Luca Daniel. Towards fast computation of certified robustness for relu networks. *arXiv preprint arXiv:1804.09699*, 2018.

[41] Eric Wong and J Zico Kolter. Provable defenses against adversarial examples via the convex outer adversarial polytope. *arXiv preprint arXiv:1711.00851*, 2018.

[42] 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*, pages 371–372, 2014.

[43] Álvaro Parafita Martínez and Jordi Vitrià Marca. Explaining visual models by causal attribution. In *2019 IEEE/CVF International Conference on Computer Vision Workshops*, pages 4167–4175, Seoul, Korea (South), October 27-28, 2019, 2019. IEEE, ICCV Workshops 2019.