Towards an Effective and Interpretable Refinement for DNN Verification

Jiaying Li[1], Guangdong Bai[2], Pham Hong Long[3] and Jun Sun[3]
[1]Microsoft, China
[2]The University of Queensland, Australia
[3]Singapore Management University, Singapore
lijiaying1989@gmail.com, g.bai@uq.edu.au, longph1989@gmail.com, junsun@smu.edu.sg

Abstract—Recently, several abstraction refinement techniques have been proposed to improve the verification precision for deep neural networks (DNNs). However, these techniques usually take many iterations to refine an abstraction and each refinement decision is hard to interpret, thus hindering their analysis, reasoning and optimization.

The goal of this work is to devise for DNN verification a new refinement technique that is both effective and interpretable, allowing analyst to understand why and how each refinement decision is made. Towards this goal, we leverage the interpretable nature of "debugging" processes and formulate the verification refinement problem as a debugging problem. In particular, we propose SURGEON, an automatic abstraction debugging approach for DNN verification. Specifically, SURGEON refines a failed DNN verification procedure iteratively and in each iteration, it first identifies the root cause of the failure and then heuristically generates fixes according to abstract transformers. We have implemented SURGEON in a prototype on the top of DeepZ and evaluated it using a set of local robustness verification problems. The experimental result shows our approach not only can improve the precision of prior verification approaches but is more effective than existing refinement techniques.

Keywords–neural network verification; abstraction refinement; abstraction debugging

1. INTRODUCTION

With the exceptional performance of deep neural networks (DNNs) in solving many challenging real world problems, recent years have witnessed their increasing integration into a wide spectrum of applications, ranging from image recognition [1], [2], [3], fraud detection [4], [5] to machine translation [6], [7]. However, their wide adoption into safety critical systems, such as self-driving cars [8], [9] and medical diagnosis [10], [11], is still in the early stage, since most DNNs still lack correctness guarantees. Ideally, DNNs should be formally verified before being deployed into critical scenarios.

In the last few years, DNN verification has received much attention from the academia. Researchers proposed multiple approaches [12], [13], [14], [15], [16], [17] to certify the correctness of DNNs. Except for precise methods which can only handle tiny neural networks, most of these efforts usually approximate DNN behaviors and trade off between analysis precision and scalability. Afterward, several refinement techniques [18], [19], [20], [21] were proposed to further enhance the verification precision. The basic idea is to split one problem into several sub-problems and then analyze each of them separately. These techniques differ in the way how the split is performed (see Sect. 6). Here, we briefly illustrate the refinement process using a simple DNN N in Fig. 1. We take DeepZ [16], an abstraction-based verification approach, as the primary DNN verifier and ReluVal [18], a refinement technique that bisects input features according to interval gradients, as the refinement method.

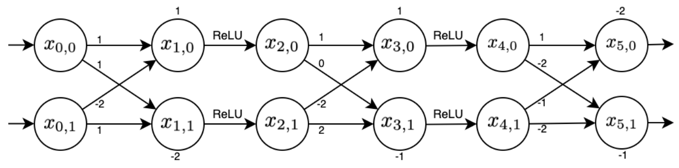


Figure 1: A $ReLU$ network, where the values on edges represent coefficients of weight matrix W and the values beside each node are biases b.

Suppose our task is to certify N is robust around an input $(0, 0)$ with perturbation range $\epsilon = 1$. That is, N yields same prediction for any input $x \in [-1, 1] \times [-1, 1]$. Initially, we employ DeepZ to verify the property, but the final abstraction fails to prove it. Then a refinement process is invoked, which is visualized in Fig. 2. In particular, after $DeepZ$ fails to verify the property, we bisect the input region with its second dimension, and consequently obtain two sub-regions, i.e. $[-1, 1]\times[-1, 0]$ and $[-1, 1]\times[0, 1]$. Next, DeepZ is further employed to prove the property for these two regions. It succeeds proving the robustness property for the first region, but fails for the second one. Afterwards, we perform the second refinement to bisect that region. This procedure continues, until all the splited regions get verified. As shown in Fig. 2, it takes four refinement actions to prove the robustness property for the perturbation region.

Although the property is verified, the whole refinement process is ineffective and hard-to-interpret, and few insights can be drawn from it. On the one hand, these techniques may take many more than needed iterations to refine an abstraction in order to improve the final verification precision. On the other hand, it is challenging for humans to interpret the refinement decisions, i.e. understanding the reason and the consequence of each refinement decision, which hinders the analysis, debugging, optimization and generalization of these refinement techniques.

Specifically, the refinement procedure in Fig. 2 can be op-

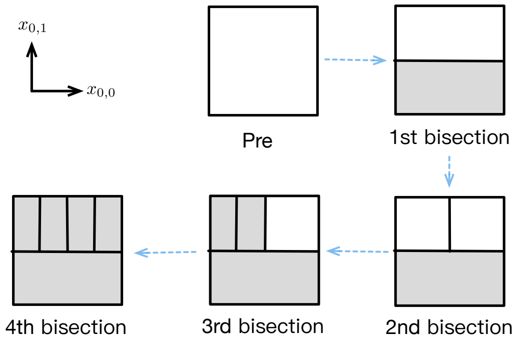


Figure 2: Refinement through input feature bisection, guided by interval gradients. The box Pre denotes the allowed perturbation region (i.e. $[-1, 1] \times [-1, 1]$), the shadowed boxes denote the regions that can be directly verified by DeepZ, and the null boxes denote regions to be verified. It takes 4 bisections before proving the robustness for Pre.

timized; if we split the abstraction for neuron $x_{1,0}$ at point 0, the two yielded sub-problems can be verified by DeepZ and thus the property is proved. More importantly, it is the transformer which takes the aforementioned abstraction as input that first introduces unsafe states into the analysis and thus splitting that abstraction makes more sense. However, previous refinement techniques provide almost no clue for identifying this "optimum".

Different from existing works, this work aims at obtaining a verification refinement technique that is both effective and interpretable. The main insight is to leverage the "interpretable" nature of debugging processes and rephrase abstraction refinement for DNN verification as a debugging problem, which we call *abstraction debugging*.

Inspired by program debugging practice, we further design SURGEON, an automatic abstraction debugging approach for refining DNN verification. Given a failed DNN verification procedure, SURGEON refines it through an iterative debugging process. In each iteration, it first attempts to locate the root cause of the failure, i.e. the earliest transformer that introduces unsafe states into the analysis. Precisely pinpointing that transformer is costly as it requires exact reasoning about DNNs. Thus, SURGEON employs optimization techniques to identify the likely root cause in a cost-effective way (see Sec. 4-B).

Next, our approach locates the corresponding transformer for fixing (see Sec. 4-C). It heuristically finds the top k impactful dimensions in the input abstraction and append to the abstract transformer fixing operators which refine an abstraction by splitting. It should be noted that, heuristics themselves do not render a procedure hard-to-interpret. Instead, good heuristics can make an refinement action more effective and thus benefits the whole refinement procedure.

One non-negligible problem of the above procedure is that our root cause identification method is generally imprecise. It may miss the real root cause transformer, in which case, SURGEON would perform the refinement in the wrong place. To overcome this problem, we further make SURGEON configurable, allowing it to trade interpretability to certain degree for the effectiveness. In particular, once identifying the likely root cause transformer, SURGEON may choose an earlier transformer as the fixing site, in order to mitigate the imprecise identification issue.

We have realized SURGEON on top of DeepZ and evaluated it using local robustness verification problems. The results show SURGEON can improve the precision of DeepZ and is more effective than existing refinement techniques.

In all, this paper makes the following contributions:

- We leverage the "interpretable" nature of debugging processes and rephrase the refinement problem for DNN verification as a debugging problem.
- We propose SURGEON, an automatic abstraction debugging approach for DNN verification. Through leveraging adversarial sampling methods and program fixing techniques, SURGEON is effective in refining DNN verification and each refinement decision is interpretable.
- We have implemented SURGEON in a prototype and evaluated it using a set of local robustness verification problems. The results confirm the precision improvement and effectiveness of our approach.

1.1 Organization.

The rest of this paper is structured as follows. Sect. 2 presents the backgrounds and briefs our approach. Then, Sect. 3 formulates the problem and Sect. 4 details our approach. Sect. 5 shows our evaluations. Finally, Sect. 6 surveys related research and Sect. 7 concludes.

2. BACKGROUNDS AND APPROACH OVERVIEW

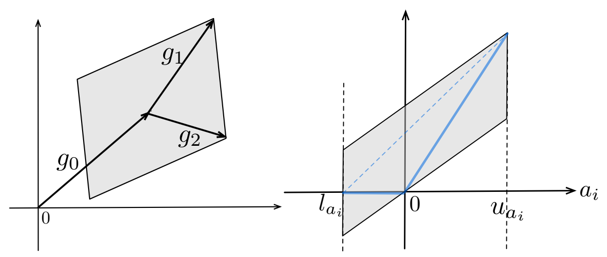
This section gives a brief introduction to the backgrounds and presents an overview of our approach with an example.

2.1 Neural Networks

Deep neural networks (DNNs) are functions that are usually organized in a layered structure. Mathematically, DNN $N = f_{n-1} \circ f_{n-2} \circ \cdots \circ f_0$ is a composite function where f_i is either an affine function, i.e. $f(x) = Wx + b$, or an activation function $\sigma(x)$ that applies in an element-wise manner. Regarding classification problems, the outputs of DNNs can be converted into categorical numbers, representing the prediction categories. More formally, DNN N classifies an input x to category c if $c = argmax(N(x))$. For simplicity, we also write N_r to denote $f_{n-1} \circ \cdots \circ f_r$.

Example 1. *Fig. 1 shows a DNN that takes rectified linear unit (ReLU), i.e.* $ReLU(x) = max(0, x)$, *as the activation function. Given an input* x_0, *the network first transforms it via an affine function, i.e.* $x_1 = (x_{0,0} - 2 \times x_{0,1} + 1,\ x_{0,0} + x_{0,1} - 2)$, *after which a ReLU activation function is applied, i.e.* $x_2 = ReLU(x_1)$. *Then,* x_2 *will be further transformed in a layer-wised manner until the output layer.* □

Well-trained DNNs have achieved exceptional performances on many problems, such as Go playing [22], [23], protein folding [24], but recent studies [25], [26] show that they are



(a) A zonotope abstraction. (b) Typical ReLU transformer.

Figure 3: DeepZ abstract domain and transformer.

subject to security issues. For instance, they can be easily fooled by adversarial examples: inputs that are similar to natural inputs but classified incorrectly by DNNs.

To be resilient to adversarial attacks, a DNN must be (locally) robust, meaning it draws similar predictions for similar inputs. Formally, we can specify an end-to-end correctness property for DNNs as a triple, i.e. $\{Pre\}N\{Post\}$, where Pre, $Post$ are the precondition and the postcondition, and the triple is valid if Pre holds, executing N establishes $Post$. Thus, we write local robustness as $\{||x - x'||_p \leq \epsilon\}N\{argmax(N(x)) = argmax(N(x'))\}$, where $||\cdot||_p$ denotes the p-norm distance, x is a given input and ϵ is the perturbation range. Due to the high complexity and non-convexity nature in the DNN computation, formally checking the robust property is a challenging problem.

2.2 Abstraction-based Verification

Most verification approaches analyze DNNs through approximation and a line of these approaches is based on abstract interpretation, a methodology that has been well studied in program verification field. Formally, abstract interpretation considers a tuple $(A, C, \alpha, \gamma, F, F^{\#})$ where C and A denote the concrete domain and abstract domain, (α, γ) are a Galois connection that gives abstraction and concretization relations between the two domains, F and $F^{\#}$ are concrete transformers and abstract transformers.

Given a triple $\{Pre\}N\{Post\}$, the analysis by abstract interpretation yields *an abstraction trace* $T = (a_0, a_1, \cdots, a_n)$ where $a_0 = \alpha(Pre)$, $a_i = f^{\#}_{i-1}(a_{i-1})$ for $i = 1, 2, \cdots, n$. The whole trace establish an abstraction for the network, while we say a_i is an abstraction for layer i. If $a_n \not\models Post$, the verification fails and we call T *a failed abstraction trace*. Otherwise, the verification succeeds and the triple is proved correct. Next, we introduce DeepZ , a typical abstract interpretation approach for DNN verification , which is the main focus in this paper.

2.2.1 Abstract Domain

DeepZ is a neural network verification approach built on zonotope abstractions. A zonotope in R^d space is defined as $a = \sum_{k=1}^{m} v_k \times g_k + g_0$, where $v_k \in [-1, 1]$ are symbolic variables, g_k are constant vectors in R^d space. In Fig. 3a, we show a zonotope abstraction in R^2 space on the left.

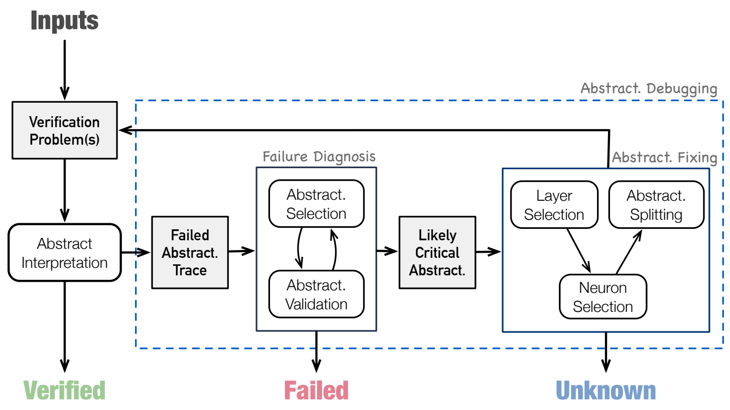


Figure 4: The overall workflow of SURGEON.

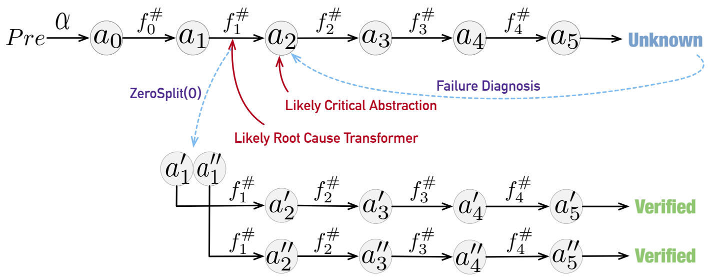


Figure 5: An illustrative refinement procedure via SURGEON.

2.2.2 Abstract Transformer

The zonotope abstraction can capture affine transformations precisely, but over-approximation is required when applied to $ReLU$ transformations. Given a zonotope a and a dimension i, DeepZ first checks whether $a_i \geq 0$ or $a_i < 0$ holds. If yes, then $ReLU$ transformation degrades into an affine transformation. Otherwise, DeepZ uses a parallelogram with the smallest area to approximate the $ReLU$ curve, as shown in Fig. 3b.

Example 2. *We take DeepZ to analyze the example verification task in Sect. 1. We first apply the abstraction function α on the input range and get the initial abstraction*

$$a_0 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \times [v_1, v_2]^T.$$

Then we apply affine$^{\#}$ transformer on a_0 and obtain

$$a_1 = \begin{bmatrix} 1 & -2 \\ 1 & 1 \end{bmatrix} \times a_0 + \begin{bmatrix} 1 \\ -2 \end{bmatrix} = \begin{bmatrix} 1 & -2 \\ 1 & 1 \end{bmatrix} \times [v_1, v_2]^T + \begin{bmatrix} 1 \\ -2 \end{bmatrix},$$

and the two dimensions can take values in range $[-2, 4] \times [-4, 0]$. Next, we apply ReLU$^{\#}$ transformer and obtain[1]

$$a_2 = \begin{bmatrix} 0.667 & -1.333 & 0.667 \\ 0 & 0 & 0 \end{bmatrix} \times [v_1, v_2, v_3]^T + \begin{bmatrix} 1.333 \\ 0 \end{bmatrix}.$$

By continuing this process, finally we have

$$a_5 = \begin{bmatrix} 0.625 & -1.25 & 0.625 & 0 & 1.256 \\ -1.25 & 2.5 & -1.25 & 0 & -0.313 \end{bmatrix} \times [v_{1,\cdots,5}]^T + \begin{bmatrix} 0.344 \\ -5.688 \end{bmatrix}.$$

Local robustness property requires $x_{5,0} > x_{5,1}$ to be hold for any concrete execution. But we cannot establish $a_{5,0} > a_{5,1}$ here and thus DeepZ fails to prove the property. □

[1]We round off all coefficients to the nearest thousandth for simplicity.

2.3 Abstraction Refinement

Due to the over-approximation, abstraction-based approaches suffer from false negative issues, i.e. they may fail to prove properties that otherwise hold. For instance, DeepZ fails to prove the robustness property which indeed hold (interval analysis [27] can indeed prove the property). To resolve the issue, the abstractions must be refined so as to eliminate unsafe states from the analysis.

Recently, multiple techniques [18], [19], [20], [21] have been proposed to refine DNN verification. The idea is to split an abstraction into multiple small sub-abstractions and analyze each of them separately. The original property is verified if it is verified on all the sub-abstractions. While boosting the analysis precision, these techniques are usually ineffective and hard to interpret, i.e. they take more refinement iterations than needed to successfully verify a property and it is very challenging for humans to understand each refinement decision, not to say to analyze, modify or even improve these techniques.

2.3.1 Overview of SURGEON

In this work, we propose SURGEON, an effective and interpretable refinement approach for DNN verification. The core insight is to leverage the "interpretable" nature of debugging processes and rephrase the verification refinement problem of DNNs as a debugging problem. Now, we brief how SURGEON works. As shown in Fig. 4, once an abstraction-based method fails to prove a property, SURGEON starts to refine the abstraction through debugging the failed abstraction trace in an iterative manner. For each iteration, it first attempts to identify the root cause via locating the likely critical abstraction, and then selects the right abstract transformer for fixing. Below, we take the failed verification procedure in Example 2 to briefly illustrate our approach.

Example 3. *Fig. 5 visualizes the entire abstraction refinement procedure by our approach. Initially, after DeepZ fails to prove the property, we collect a failed verification trace as shown on the top of the figure. Next, our approach attempts to identify the root cause transformer of this failure. Through validation, it finds out abstraction a_1 is safe while abstraction a_2 is unsafe, indicting it is the transformer $f_1^{\#}$ that introduces unsafe states into the analysis. Therefore, our approach starts to fix $f_1^{\#}$. The fixing process works by inserting a split operation before the original transformation. That is, before applying $f_1^{\#}$, our approach first performs a split operation on a_1, so that the over-approximation introduced by $f_1^{\#}$ can be eliminated. With some heuristics, it chooses to split a_1 on its first dimension at point 0, and consequently yields two sub-abstractions, i.e. a_1' and a_1'', respectively. Then we construct two sub-problems, i.e. $\{a_1'\}N_1\{Post\}$ and $\{a_1''\}N_1\{Post\}$, for verification. Next, DeepZ is further employed and finally both problems get verified. Therefore, the original problem gets verified and* SURGEON *reports* `'VERIFIED'` *as the result.* □

3. TECHNICAL FORMULATION

This part revisits program debugging and formulates the verification refinement problem for DNNs as a debugging problem.

3.1 Program Debugging

Program debugging is an important and large topic that has been well-studied by the industrial and research community. This part focuses on the "interpretable" [2] nature of debugging.

3.1.1 The "Interpretable" Nature

Program debugging is usually an iterative process and each iteration consists two sub-procedures: *failure diagnosis*, aiming at locating root causes of the failure, and *fix generation*, aiming at generating fixing action for the failure.

Let us recall how developers debug program in practice. To debug a program, developers usually need to inspect, execute, trace or profile the program either manually or in a tool-aided way. Then they will guess the root cause of the failure and validate the guess afterward. Once the guess is verified, they can start to patch the program with their expertise; otherwise, they need to take a new round and form new guesses. The whole debugging procedure greatly involves humans' efforts and is trustworthy only if developers have made proper decisions at each step of the process. Moreover, once a fix is submitted, the industrial practice would launch a reviewing process, aiming at evaluating the fix in term of its simplicity, effectiveness, and interpretability. Such reviewing process may requires developers to further explain the root cause of the failure and why their fix can address the failure. intrinsically requiring the debugging procedure to be effective and interpretable. Otherwise, a fix can be hardly accepted.

Debugging is essentially reasoning about the causality over the program and the root cause indeed plays a crucial role in program debugging. In fact, whether we can identify the root cause of the failure matters to the effectiveness and the interpretability of a debugging procedure. Without identifying the root cause, a fix action is either blinded trials or heuristics-guided, which is (1) ineffective, providing no guarantee in mitigating or resolving the program failure, and (2) hard-to-interpret, hindering others from understanding why and how the fix is performed.

3.1.2 Formulation

Formally, we can model a program as a dependency graph where the semantics is associated with each node that denote the program statement(s).

Definition 1 (Dependency Graph)**.** *A program S with a desirable property ϕ that fails under certain conditions is a* dependency graph *$G = (M_s, M_f, E, entry, error)$, where M_s are a set of nodes representing program states, N_f are a set of nodes representing program statements* [3]*, $E \subseteq (M_f \cup M_s) \times M_s$ are directed edges that represent the dependency relation between*

[2]Interpretable issues have been discussed in various scenarios, but a formal and widely accepted definition is yet to establish. A common yet informal definition is: Interpretability is the degree to which a human can understand the cause of a decision [28].

[3]In this work, we model each program statement as one graph node. In general, we may use one node to represent a sequence of program statements, a.k.a. code blocks or code snippets.

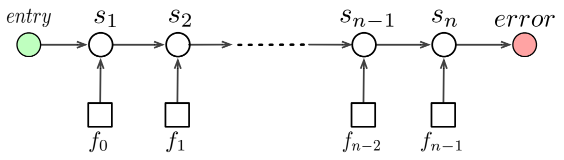


Figure 6: The dependency graph for a straight-line program.

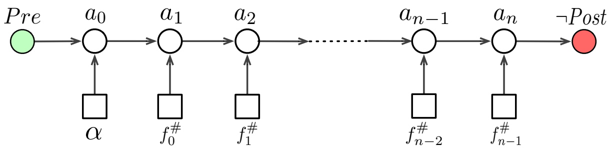


Figure 7: The dependency graph for an abstraction analysis.

each node, $entry \in M_s$ and $error \in M_s$ denote the entry states and the error states, respectively.

The graph provides a way to visualize the dependency relation between different program states and between program states and statements, and it allows us to reason about how program states evolves over the execution of each program statement. For example, a straight-line program $S := f_0; f_1; f_2; \cdots; f_{n-1}$ can be modeled as a dependency graph shown in Fig. 6. The dependency graph can ease us to identify problematic statements that contributes to the program failure. Regarding the same example, assume node s_1 is correct. If we observe there is one concrete execution starting from a state in s_2 finally reaching $error$, then we know for sure that statement f_1 is problematic.

In order to resolve the program failure, its dependency graph should be modified in some way, exactly what program debugging procedure does. While constructing a complete different program to resolve the failure is always possible, in program debugging we usually consider small modifications and avoids big changes as much as possible. In particular, it is not allowed to directly modify program states N_s, and for simplicity, we further restrict all the modification within a set $\Delta := \{\delta_0, \delta_1, \cdots, \delta_m\}$ where $\delta_i : f_j \to f'_j$ is a fixing operator that is applicable to some program statement f_j. For example, a program statement "`c = a + b`" can be converted into "`c = a − b`" through a fixing operator that maps "+" to "−". Therefore, we have:

Definition 2 (Debugging Procedure)**.** *Let G be a dependency graph for program S with property ϕ, and Δ be a set of fixing operators. Assume P is a procedure that takes S and ϕ as input, and outputs a sequence of fixing actions $p_0, p_1, \cdots$, where p_i is a pair (f_j, δ_k) wherein f_j is the program statement to fix and $\delta_k \in \Delta$ is the fixing operator to apply. We call P a* debugging procedure *if applying $p_0, p_1, \cdots$ to S can mitigate or fix the failure.*

Program debugging itself does not provide effective and interpretable solution. Actually, debugging large programs is a daunting task that often needs to be done manually, especially when the dependency graph is complicated and the size of the fixing operator set is very large, if not infinite.

3.2 Abstraction Debugging

Different from existing refinement techniques, in this work we rephrase abstraction refinement as the debugging problem for a special program, which is the sequence of applied abstract transformers during DNN verification.

Specifically, given a triple $\{Pre\}N\{Post\}$, when an abstract interpretation approach fails to prove it, a straight-line program $S := \alpha; f_0^{\#}; \cdots; f_{n-1}^{\#}$ can be automatically constructed. Therefore, we can reduce an abstraction refinement problem to a debugging problem, termed "abstraction debugging" in this paper. Although such refinement-debugging reduction is straightforward, the debugging problem itself does not provide effective and interpretable solutions.

Similar with program debugging, we first model S as a dependency graph $G = (M_s, M_f, E, entry, error)$, where M_s is a set of abstractions appeared in the failed verification trace T, M_f is the a set of abstract transformers defined by the abstract interpretation framework, $E \subseteq (M_f \cup M_s) \times M_s$ denotes the dependency relations, $entry$ and $error$ denote Pre and $\neg Post$, respectively. One typical dependency graph is shown in Figure 7.

Once having a dependency graph representing the failed verification procedure, the next step is to figure out which abstract transformer node to fix, and what fixing operators we can apply. In other words, we need to identify the root cause of the verification failure. However, it might be the case that several abstract transformer contributing to the verification collectively, and therefore fixing any of them cannot rescue the verification from failure. Under this condition, a practical and effective strategy is to start the fixing from the earliest transformer that introduce unsafe states into the analysis. Thus, we take that transformer as the root cause of the failure, and in the following, we formally define the root cause together with several concepts.

Definition 3 (Unsafe Abstraction)**.** *Let $\{Pre\}N\{Post\}$ be the triple to verify and $T = (a_0, a_1, \cdots, a_n)$ be a failed abstraction trace. a_i is* unsafe *if there exists $x_i \in \gamma(a_i)$ such that $N_i(x_i) \not\models Post$.*

Intuitively, an abstraction is unsafe as its concretization contains some data points that can lead to the target property violation. Note that, Pre here can be the input constraint for any layer. For example, during the abstraction refinement, we may be interested in a partial DNN such as N_2, then Pre should be a_2 [4]. Apparently, unsafe abstractions must be refined before the target property can be verified. More precisely, given a failed abstraction trace, we would like to identify the "critical abstraction", an unsafe abstraction that indicts the root cause of the failure.

[4]If we consider a numerical abstraction a as a floating point constraint, then set $\{x|x \in a\}$ and set $\{x|x \in \gamma(a)\}$, where x is a floating point number, are equivalent. Therefore,we use $\gamma(a)$ and a interchangeably in this paper.

Definition 4 (Critical Abstraction). *Let $\{Pre\}N\{Post\}$ be the problem to verify and $T = (a_0, a_1, \cdots, a_n)$ be a failed abstraction trace. a_c is the* critical abstraction *if and only if:*
(1) for all $0 \le i < c$, we have a_i is safe;
(2) for all $c \le i \le n$, we have a_i is unsafe.

With above, we define the root cause transformer of the verification failure now.

Definition 5 (Root Cause Transformer). *Let $\{Pre\}N\{Post\}$ be the problem to verify, $T = (a_0, a_1, \cdots, a_n)$ be a failed abstraction trace, a_c be the critical abstraction. We say $f^{\#}_{c-1}$ is the root cause transformer of the failure.*

Here, we take $f^{\#}_{c-1}$ as the root cause of the failure, because it is the earliest abstract transformer that introduces unsafe states into the analysis. Moreover, $f^{\#}_{c-1}$ should be the first transformer to fix. Otherwise, if we fix latter transformers, the fix can not be effective; if we fix earlier transformers, the fix can be hard-to-interpret.
Up to now, we have formulated the abstraction refinement problem for DNN verification. In particular, we have reduced abstraction refinement to an abstraction debugging problem, and reduced the root cause identification to the problem of locating the critical abstraction. In the next section, we present our approach to abstraction debugging.

4. Automatic Abstraction Debugging

Inspired by some techniques used in program debugging, we devise an automatic abstraction debugging approach to refine DNN verification. This section presents the detail.

4.1 Overall Algorithm

The overall algorithm of our approach is shown in Algo. 1. It takes a verification problem, i.e. $\{a_i\}N_i\{Post\}$, and a failed abstraction trace T as input, and outputs several sub-problems to further verify. Note that, when this procedure is invoked for the first time, i.e., i is equal to 0, we have $a_i = \alpha(Pre), N_i = N$. The main procedure works as follows. First, it locates the likely critical abstraction through a binary search, implemented by the loop from Line 2 to Line 8. During each step of the search, it invokes procedure $AbstractValidate$ to check the safety of an abstraction. Once the likely critical abstraction $a_{\tilde{c}}$ is identified, procedure $AbstractFix$ is adopted to decide the right transformer to fix and perform the actual split. Finally, the resulting sub-abstractions are encoded into verification problems and the control returns to the original verification process. Note that, although our approach is quite general, some operations might need to be tuned for specific abstract interpretation approach. In the following, we take DeepZ for example to present how abstraction validation and abstraction fixing are performed in our approach.

4.2 Abstraction Validation

In principle, deciding whether an abstraction is safe requires precisely reasoning about (partial) DNNs, which is of great challenge. Thus, instead of seeking for perfect methods for abstraction validation, our goal here is to perform the validation in a cheap yet effective way.

Algorithm 1: Abstraction Debugging Algorithm

```
Inputs : problem $P = \{a_i\}N_i\{Post\}$, failed verification
         trace $T = (a_i, \cdots, a_n)$.
Output: a set of sub-problems $Pset$.
1  $begin, end \leftarrow i, n$
2  while $begin < end$ do
3      $mid \leftarrow (begin + end + 1)/2$
       // check whether $a_{mid}$ is safe or not.
4      $safe \leftarrow AbstractValidate(a_{mid},\ N_{mid},\ Post)$
5      if $safe$ then
6          $begin \leftarrow mid + 1$
7      else
8          $end \leftarrow mid$
9  $\tilde{c} \leftarrow mid$
   // now $a_{\tilde{c}}$ is the likely critical abstraction.
10 $t, aset \leftarrow AbstractFix(P,\ T,\ \tilde{c})$
   // construct new problems from resulting abstractions.
11 $Pset \leftarrow [\ ]$
12 foreach abstraction $a$ in $aset$ do
13     $Pset.append(\{a\}N_t\{Post\})$
14 return $Pset$
```

The basic idea is to sample data points within the region specified by the abstraction, i.e. $\gamma(a)$, and check whether any of them leads to property violation. Note that trivial sampling based on the value range is ineffective since *i)* an abstraction usually specifies a geometrically complicated region where different dimensions may have some dependency; and *ii)* a large number of valid samples is needed in order to build the confidence for an abstraction. Therefore, we decide to sample data points as follows.
First, our method samples seed data points within the abstraction. Our observation is, elements in widely-used numerical abstract domains (powersets are not considered here) can often be represented using generation functions over some meta-variables. For example, in the zonotope abstract domain, we can define a function as $Gen(v) = \sum_{k=1}^{m} v_k \times g_k + g_0$, where v_k are the meta-variables (the so-called symbolic variables in the literature). In this case, any data point $x \in \gamma(a)$ can be reached through $Gen(v^*)$ where v^* is a concrete valuation of v. Since meta-variables are usually constrained within a high dimensional box that is much more regular than the resulting abstraction, we can get valid x in two steps: first sample valid data point v^*, and then compute x through Gen.
Then, our method searches for counterexamples through pushing seed data points towards the unsafe zone. Inspired by adversarial generation [29], we formulate an optimisation problem and leverage the gradient information to search the points. 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 towards the unsafe zone through decreasing

Algorithm 2: Abstraction Validation Algorithm

```
Inputs : $a_i$, partial DNN $N_i$, $Post$.
Configs: optimization parameters: $u_1, u_2, step\_size$
Output: $True$, if $a_i$ is very likely a safe abstraction;
        $False$, if $a_i$ is an unsafe abstraction.
1  Let $Gen$ be the generation function of $a_i$
2  while less than $u_1$ iterations do
3      Sample valid $v^*$ within the value range of $v$
4      while less than $u_2$ iterations do
5          $x_i \leftarrow Gen(v^*)$
6          if $N_i(x_i) \not\models Post$ then
               // counterexample found.
7              if $i = 0$ then
                   // counterexample exists at input layer.
8                  report 'Failed' with $x_i$ and exit
9              return $False$
10         $grad \leftarrow \frac{\partial Loss(x_i, N_i, Post)}{\partial x_i} \circ \frac{\partial Gen}{\partial v}(v^*)$
           // update the value of $v^*$ and do the projection.
11         $v^* \leftarrow v^* - step\_size \times sgn(grad)$
12         Clip $v^*$ to fit the value range for $v$
13 return $True$
```

the loss value.

We show the detail in Algo. 2. The loop from line 2 to line 12 allows us to search from multiple random seeds until a timeout occurs. At line 3, we generate a random seed v^* and compute the data point x_i through function Gen. The loop from line 4 to line 12 then navigates through the space of x_i iteratively. Here we adopt the projected gradient descent (PGD) [30], a standard method for large-scale constrained optimization, to optimize the loss function. At line 6 we check whether x_i leads to property violation. If yes, then a_i must be unsafe and we further check whether a_i is the abstraction for the input layer. Otherwise, we follow the gradient to search through the space. That is, at line 11, we modify x_i according to a loss function.

Before closing this part, we present how the loss function is constructed. For the local robustness analysis, a simple loss function can be defined as:

$$Loss(x_i, N_i, Post) \quad = \quad N_i(x_i)_k - \max_{j \neq k}(N_i(x_i)_j)$$

where k is the desired class specified by $Post$. Apparently, decreasing the value of the loss function navigates the input x_i towards the property-violating zone (i.e. unsafe zone). Once a negative loss value is reached, the optimization procedure identifies a counterexample for the original problem. It should be noted such methods are usually incomplete; it may fail to find out counterexamples which indeed exist.

4.3 Abstraction Fixing

The primary means of abstraction fixing in this work is through splitting. Although it is not hard to split an abstraction,

Algorithm 3: Abstraction Fixing Algorithm

```
Inputs : problem $P = \{a_i\}N_i\{Post\}$, failed trace $T$,
         index of the likely critical abstraction $\tilde{c}$.
Configs: maximum number of splits per fix $k$.
Output: fixing layer $\tilde{r}$, abstractions $aset$.
   // locate the likely root cause transformer $f_{\tilde{r}}^{\#}$ according to $\tilde{c}$.
1  $\tilde{r} \leftarrow \tilde{c} - 1$
2  $dims \leftarrow DimensionFilter(a_{\tilde{r}})$
   // find top $k$ dimensions with largest influences, and split.
3  $kdims \leftarrow InfluenceRank(a_{\tilde{r}},\ dims,\ k)$
4  if $kims$ is empty then
5      report 'UNKNOWN' and exit
6  $aset \leftarrow zSplit(kdims)(a_{\tilde{r}})$
7  return $\tilde{r},\ aset$
```

choosing which abstraction to split and how to split it is critical to the effectiveness of our fixing procedure.

As we know, the $affine^{\#}$ transformer in DeepZ is exact, while the $ReLU^{\#}$ transformer is inexact. A $ReLU^{\#}$ transformation over-approximates the $ReLU$ behavior when any dimension of its input abstraction's value range spans across zero, and thus can introduce unsafe states into the analysis. Under this condition, the abstraction must be refinement in order to eliminate the unsafe states. While performing a complex cross-dimensional split is possible and can be more effective, SURGEON takes one simple yet easy-to-interpret choice, splitting the input abstraction on that dimension at point 0. More formally, if we use $zSplit(d_0)(a)$ to denote the action that splits abstraction a on the dimension d_0 at 0, then we can a fixing operator for DeepZ as follows

$$\delta(ReLU^{\#}) = ReLU^{\#} \circ zSplit(d_0)$$

where d_0 is any dimension on which the input abstraction's value range spans across 0.

Sometimes splitting an abstraction once cannot eliminate all the unsafe states introduced by the $ReLU^{\#}$ transformer, and thus we need split an abstraction multiple times (on different dimensions). For this reason, we further define

$$zSplit(d_0, \cdots, d_p) = zSplit(d_p) \circ zSplit(d_0, \cdots, d_{p-1})$$

where $p > 0$ and $d_0, \cdots, d_p$ are distinct dimensions of the input abstraction. So we have a series of fixing operators

$$\delta(ReLU^{\#} \circ zSplit(d_0, \cdots, d_{p-1})) = ReLU^{\#} \circ zSplit(d_0, \cdots, d_p).$$

At this point, we are ready to establish the fixing operator set used in SURGEON:

$$\Delta = \{\delta(ReLU^{\#}), \delta(ReLU^{\#} \circ zSplit(d_0, \cdots, d_{p-1})\ \forall\, p > 0\}.$$

Based on the above discussion, now let us introduce the abstraction fixing algorithm in our approach, as shown in Algo. 3. Given $\tilde{c}$ is the index of likely critical abstraction, it calculates the likely root cause transformer $f_{\tilde{r}}$ and then filters out some dimensions of $a_{\tilde{r}}$ (i.e. input abstraction to $f^{\#}_{\tilde{r}}$) where no over-approximation is introduced. Function $InfluenceRank$ is employed to rank the left dimensions in which the top k are selected for splitting. Finally, after $zSplit$ perform the actual splitting on $a_{\tilde{r}}$, the algorithm returns.

Lastly, we discuss how $InfluenceAnalysis$ works. Its goal is to rank different dimensions of an abstraction based on their impacts on the output layer of the DNN. SURGEON borrows the idea from ReluVal [18] and uses the interval gradient as an indicator of the impacts, where a larger interval gradient value suggests that dimension of the abstraction has a greater influence towards changing the output, and thus splitting on it would improve the overall analysis precision more effectively.

4.4 Remedy to Imperfect Validation

Up to here, we have presented core techniques for automatic abstraction debugging. It should be noted that the effectiveness of our approach can be threatened by the imperfectness of our abstraction validating procedure. Specifically, SURGEON validates an abstraction through optimization methods, which are cost-effective yet incomplete, i.e. it may report an unsafe abstraction safe, in which case, SURGEON fails to identify the actual root causes. Worse still, the failure cannot be fully resolved with the later refinement decisions.

To overcome this problem, we make SURGEON configurable, allowing it to recover from diagnosis failure to certain degrees. In specific, once identified the likely root cause, SURGEON may refine the domain based on some earlier abstraction, considering the abstraction validating procedure may fail to detect the actual root causes. We should note this remedy could decrease the interpretability of our approach, as it may perform refinement action based on safe abstractions.

4.5 Soundness

The primary means of verifying DNN in SURGEON is abstraction and thus we focus on how abstractions are manipulated in the entire procedure. In fact, abstractions can be only manipulated in three ways: (1) created by abstracting Pre; (2) computed through applying abstract transformers to existing abstractions and (3) reconstructed on the refined abstract domain or computed through applying abstract transformers to new abstractions. The soundness of both (1) and (2) is guaranteed by base verification approaches. Regarding (3), each disjunctive component of the abstraction is obtained through (2) and thus the soundness is also guaranteed. Furthermore, the soundness of the disjunctive operation is guaranteed. Therefore, the soundness of SURGEON is proved.

5. EVALUATION

To evaluate our proposal, we have conducted an extensive set of experiments to answer the following research questions. In this section, we present the detail.

RQ1: Can SURGEON improve the verification precision of DeepZ?
RQ2: How does SURGEON perform against abstraction refinement techniques in analysis precision?
RQ3: Is SURGEON more effective than abstraction refinement techniques?
RQ4: Does SURGEON scale to large DNNs?
RQ5: Does the remedy help repair abstract interpretation?

5.1 Implementation

As a proof-of-concept demonstration, we have implemented SURGEON in a prototype tool based on DeepZ. The main procedure is written in Python and it leverages ELINA library [31], [32] to perform the underlying abstraction analysis. Noted that some modification to ELINA is needed in order to support partitioning operation on numerical domain elements.

a) Parallelism.
Our approach can be easily parallelized, as the multiple iterations of the optimization for abstraction validation and different sub-problem verification can be run on different threads. Yet, we did not enable parallelism in our evaluation, for the sake of making fair comparison between different techniques.

5.2 Experimental Setup

a) DNN Models.
We have trained 16 fully connected DNN models on the MNIST dataset [33] , which consists of 60000 grayscale images of handwritten digits ($0 \sim 9$), each with 28×28 pixels. The DNNs have sizes $\{3, 5, 7, 9\} \times \{5, 10, 50, 100\}$ where $N \times M$ means there are N hidden layers and each layer has M neurons. We show more detail in Table I, where the data under '*overall*' columns denote the overall precision over the testing dataset, and the data under '*first100*' columns denote how many inputs among the first 100 inputs from the training dataset are correctly classified by the DNN (These inputs were examined throughout our evaluation, as will be explained shortly.).

	M=5		M=10		M=50		M=100	
	overall	first100	overall	first100	overall	first100	overall	first100
N=3	0.8304	90	0.9359	96	0.9708	99	0.9759	98
N=5	0.8675	87	0.9247	91	0.9709	99	0.9747	100
N=7	0.7777	79	0.9333	99	0.9703	99	0.9765	99
N=9	0.7351	74	0.915	92	0.9694	99	0.9788	100

Table I: DNN models used in our evaluation.

b) Robustness Properties.
As [17], we took the first 100 inputs from the MNIST training dataset for L_∞-norm [26] distance based robustness analysis. Unlike previous literature [16], [17] where the perturbation ϵ is fixed, i.e. $\epsilon \in \{0.005, 0.01, 0.015, 0.02, 0.025, 0.03\}$, we examine the *exact verifiable robustness* (denoted as *EVR* in the following) for each method with respect to each DNN and each input during the evaluation. That is, we gradually increase the perturbation ϵ from 0.001 up to 1.0 with step size 0.001, and record the largest perturbation ϵ value that can be verified by the corresponding verification method as its *EVR*. With this fine-grained adjustment, the capacity of each method can be precisely assessed and fairly compared.

SURGEON v.s. DeepZ	M=5		M=10		M=50		M=100	
	#	$\nearrow$	#	$\nearrow$	#	$\nearrow$	#	$\nearrow$
N=3	90	35	96	59	99	62	98	47
N=5	87	35	91	66	99	44	100	8
N=7	79	10	99	45	99	17	99	5
N=9	74	19	92	37	99	11	100	2

(a)

SURGEON v.s. DeepZ^r	M=5		M=10		M=50		M=100	
	$\nearrow$	$\searrow$	$\nearrow$	$\searrow$	$\nearrow$	$\searrow$	$\nearrow$	$\searrow$
N=3	35	0	58	0	62	0	42	0
N=5	35	0	64	1	40	1	6	0
N=7	10	0	41	1	14	0	1	3
N=9	18	1	36	1	9	2	4	1

(b)

SURGEON v.s. DeepZ^g	M=5		M=10		M=50		M=100	
	$\nearrow$	$\searrow$	$\nearrow$	$\searrow$	$\nearrow$	$\searrow$	$\nearrow$	$\searrow$
N=3	34	3	59	1	57	3	41	0
N=5	28	2	64	1	35	1	5	2
N=7	9	3	36	2	8	0	2	7
N=9	15	3	34	3	6	4	3	2

(c)

Table II: Precision comparison between SURGEON with DeepZ, DeepZ^r and DeepZ^g.

c) Experimental Configuration.
All experiments are performed on a Ubuntu 16.04 machine with a 1-core CPU and a 8GB memory. By default we take 2 minutes as timeout and 256 as the maximum number of fixes for each verification problem. We set the optimization parameters as $u_1 = 8, u_2 = 400, step\_size = 0.005$, and the maximum number of splits per fix as $k = 1$.

5.3 Precision Improvement over DeepZ (RQ1)

The primary goal of our approach is to improve the verification precision of existing approaches, and this research question aims at access this goal empirically. In particular, we would like to examine the *EVR* for each input and each model, and compare whether the *EVR* of SURGEON outperform the baseline, i.e. the *EVR* of DeepZ. Note that, we should exclude the inputs that are mis-classified by a given DNN.
The result is shown in Table IIa. In the table, N and M denote the DNN structure, the data in the columns titled "#" show the number of inputs that are correctly classified by a specific DNN, and the data under in the columns titled "$\nearrow$" show the number of inputs whose *EVR* get improved by SURGEON. We can see from the table that, SURGEON can indeed improve the verification precision of DeepZ, and the ratio of the improved inputs can be up to 72% (i.e. network 5×10).

5.4 Precision v.s. Other Refinement Techniques (RQ2)

This research question focuses on the precision comparison between our approach and existing refinement techniques. To approach this question, we consider DeepZ with two refinement techniques based on input feature bisection, wherein one chooses input features randomly, denoted as DeepZ^r, and the other chooses them based on interval gradients, denoted as DeepZ^g. We ran SURGEON, DeepZ^r and DeepZ^g on all the DNNs and all the inputs, and collected the *EVR* for each of them.
The statistical result is shown in Table IIb and Table IIc, where the former shows the comparison result between SURGEON and DeepZ^r, and the latter shows the comparison result between SURGEON and DeepZ^g. The data in the columns titled "$\searrow$" show the number of inputs whose *EVR* on SURGEON are smaller than the compared technique, while the other columns hold the same meaning as before. For all the DNNs except network 7×100, SURGEON outperforms the compared refinement techniques in term of the verification precision. In fact, SURGEON works extremely well for DNNs in small sizes.

5.5 Effectiveness v.s. Other Refinement Techniques (RQ3)

This research question cares about the effectiveness of our approach. In essence, we should compare the refinement steps that are taken by different refinement techniques, in order to achieve the same precision improvement. However, given a DNN, identifying inputs whose *EVR* can be improved by all the refinement techniques is tedious. Thus we evaluate the effectiveness from a different angle. We fix the maximum refinement steps, and compare the *EVR* for different analysis methods, with respect to all the network and all the inputs. For example, given a DNN and an input x, assuming we set the maximum refinement step to be 10, then method A is more effective than method B if the *EVR* of x by A is larger than the *EVR* by B. In particular, we took $4, 8, 16, 32$, as the maximum refinements for each verification problem, and compared the *EVR* of SURGEON with DeepZ^r and DeepZ^g.
The detailed result is show in Table III and Table IV. The result shows that SURGEON is much more effective than DeepZ^r in verifying small DNNs, where $M = \{5, 10, 50\}$. When $M = 100$, they are still competitive. Such saying holds in comparing SURGEON with DeepZ^g.
However, the result shows that our approach is less effective than DeepZ^g for larger DNNs, i.e. $M = 100$. We suspect this is due to the imperfectness of abstraction validation procedure.

5.6 Scalability (RQ4)

Here, we take the data collected for the above questions, i.e. from Table II to Table V, to study the scalability of our approach. In particular, we compare the precision improvement of SURGEON over other approaches among different models. Usually, SURGEON performs much better than other methods on small DNN models. With the DNN gets larger, the experimental result shows the precision gain of SURGEON is indistinctive. (The number of inputs whose *EVR* get proved drops from more than 30 to 1 or 2.)
This is because, abstraction validation and neuron selection is quite challenging for large DNNs. With more neurons in each hidden layer, the impact of each neuron to the output layer might be subtle and dozens of neurons might need to be refined in order to fix the failure. All these suggest that it is non-trivial for our approach to scale to large DNNs. Indeed, further research is demanding in order to get an effective, interpretable and scalable approach for abstraction refinement.

Max Refinement = 8								
	M=5		M=10		M=50		M=100	
	$\nearrow$	$\searrow$	$\nearrow$	$\searrow$	$\nearrow$	$\searrow$	$\nearrow$	$\searrow$
N=3	35	0	58	0	55	0	33	0
N=5	35	0	65	1	32	0	3	0
N=7	9	0	42	0	8	0	2	3
N=9	19	0	35	0	7	1	2	2

Max Refinement = 16								
	M=5		M=10		M=50		M=100	
	$\nearrow$	$\searrow$	$\nearrow$	$\searrow$	$\nearrow$	$\searrow$	$\nearrow$	$\searrow$
N=3	36	0	58	0	61	0	36	0
N=5	36	0	66	1	36	1	4	0
N=7	10	0	43	0	12	0	2	4
N=9	18	0	35	0	9	1	2	2

Max Refinement = 32								
	M=5		M=10		M=50		M=100	
	$\nearrow$	$\searrow$	$\nearrow$	$\searrow$	$\nearrow$	$\searrow$	$\nearrow$	$\searrow$
N=3	36	0	58	0	62	0	40	0
N=5	35	0	65	1	40	1	4	0
N=7	10	0	44	0	11	0	3	4
N=9	18	1	35	1	8	2	2	2

Table III: Precision comparison between SURGEON with DeepZ^r, given the maximum number of refinement is fixed.

Max Refinement = 8								
	M=5		M=10		M=50		M=100	
	$\nearrow$	$\searrow$	$\nearrow$	$\searrow$	$\nearrow$	$\searrow$	$\nearrow$	$\searrow$
N=3	35	0	57	0	53	0	31	0
N=5	33	0	64	1	30	1	2	1
N=7	9	0	39	0	6	0	0	7
N=9	18	0	32	1	5	3	2	2

Max Refinement = 16								
	M=5		M=10		M=50		M=100	
	$\nearrow$	$\searrow$	$\nearrow$	$\searrow$	$\nearrow$	$\searrow$	$\nearrow$	$\searrow$
N=3	35	0	58	0	58	0	34	0
N=5	34	0	65	1	30	1	3	3
N=7	10	1	38	1	7	0	1	7
N=9	17	1	31	1	5	5	2	4

Max Refinement = 32								
	M=5		M=10		M=50		M=100	
	$\nearrow$	$\searrow$	$\nearrow$	$\searrow$	$\nearrow$	$\searrow$	$\nearrow$	$\searrow$
N=3	35	0	58	0	57	2	37	0
N=5	33	2	64	1	33	1	3	4
N=7	10	1	36	1	8	0	1	9
N=9	16	1	32	2	4	5	3	5

Table IV: Precision comparison between SURGEON with DeepZ^g, given the maximum number of refinement is fixed.

5.7 Impact of the Remedy on the Analysis Precision (RQ5)

In the last question, we aim at assessing whether the remedy help to repair abstract interpretation. That is, whether choosing an earlier abstraction to guide the refinement could contribute to the precision of the verification. In the following, we configure SURGEON with different δ, which measures how far the chose abstraction is from the identified root cause. For example, when $\delta = 0$, we just choose the identified root cause abstraction to guide the fixing; and when when $\delta = 1$, we choose the abstraction right before the identified root cause, i.e. a_{r-1}, to guide the fixing. In the following, we run SURGEON with different δ values (i.e. $\delta = \{0, 1, 2, 3\}$), denoted as $\delta-$SURGEON, where $\delta = 0$ denotes the baseline, and compared the number of inputs whose *EVR*s get improved.

	M=5				M=10			
	δ=0	δ=1	δ=2	δ=3	δ=0	δ=1	δ=2	δ=3
N=3	35	+1	+1	-	59	+8	+10	-
N=5	35	+4	+10	+12	66	+11	+14	+15
N=7	10	+3	+3	+3	45	+13	+14	+17
N=9	19	+1	+1	+2	37	+10	+17	+20

	M=50				M=100			
	δ=0	δ=1	δ=2	δ=3	δ=0	δ=1	δ=2	δ=3
N=3	62	+10	+22	-	47	+5	+17	-
N=5	44	+3	+11	+25	8	+0	+16	+24
N=7	17	+0	+2	+5	5	+1	+6	+13
N=9	11	+1	+4	+8	2	+0	+2	+2

Table V: Improvement of $\delta-$SURGEON over DeepZ.

The results are shown in Table V, where each column under '$\delta = \{1, 2, 3\}$' shows the number of inputs whose *EVR* get improved by $\delta-$SURGEON, compared with baseline DeepZ, i.e. the columns under '$\delta = 0$'. It shows that, as δ increases, $\delta-$SURGEON can verify more inputs with higher *EVR* values. It indicates that, given the strength of the current validation technique, choosing earlier abstraction to guide refinement can indeed help repair abstract interpretation. In specific, when δ is small, it is very likely for our validation technique to miss the critical abstraction. Consequently, SURGEON may attempt to refine the abstract domain based on a late abstraction, and thus the verification procedure cannot be corrected. However, through increasing the value of δ, we are given more chances to jump backwards and thus the corresponding refinement are more likely to resolve the failure. To the extreme case, when we ignore the critical abstraction and just refine the abstract domain based on the input feature, the abstract domain can grow too fine grained, possibly blowing up to a great many of tiny intervals, or even the whole concrete domain.

6. RELATED WORK

In this section, we survey the most related research.

6.1 DNN Verification

Several approaches have been proposed to verify DNNs. According to the analysis precision, they can be grouped into two categories: exact methods and approximation methods.

Exact methods consider the precise semantics of DNNs. For instance, [14] adopts mixed integer linear programming (MILP) to tackle the verification problem, while [12], [13] tackle the problem through SMT solving. These methods are sound and complete, but usually are costly and limited in scalability. In comparison, approximation methods choose to sacrifice precision for efficiency and scalability. For instance, the approaches in [34], [35] adopt linear approximation, whereas the approaches in [15], [16], [17] leverage numerical abstract domains to reason about DNNs. Through proper over-approximation, these methods usually could handle larger networks than exact methods.

6.2 Abstraction Refinement for DNN Verification

Recently, multiple research proposed abstraction refinement techniques to improve the analysis precision of the approximation approaches. Among them, several works split abstraction through manual crafted strategies, such as ReluVal [18], which bisects the input space with the guidance of symbolic interval gradients, and Neurify [19], which further extends the work by refining on external neurons. Later, these strategies can be gained through a data-driven learning. [20] leverages Bayesian optimization to train the domain strategy and split strategy and then apply them to facilitate DNN verification. [21] adopts GNNs to learn more complicated refinement strategies. Our work belongs to this line of research, but it is of debugging nature, which is more interpretable, allowing users to understand each refinement step of the process.

6.3 Counterexample Guided Abstraction Refinement

Counterexample Guided Abstraction Refinement (CEGAR) [36] is a generic technique that iteratively constructs and refines abstraction until a proper precision is reached. In the last decade, CEGAR has been applied to verify a variety of software systems [37], [38], [39], [40] and hardware systems [36], [41]. Here we only consider CEGAR research in program verification, as DNNs can be regarded as a new paradigm for programming.

In program verification, a line of research such as [42], [43], [36], [44], [45], [46], [47] have explored to adopt CEGAR to facilitate verification, where the core problems are to decide when and how to apply the refinement. In general, our work is closely related to CEGAR-based approaches, as the whole refinement procedure is driven by counterexamples. However, they are also very different. In addition to the different application domains, the primary usage of the counterexample in our approach is to locate the root cause site of the failure and the refinement is based on the identified root cause site.

7. Conclusion

In this paper, we leverage the "interpretable" nature of debugging processes and propose SURGEON, an effective and interpretable debugging approach for refining DNN verification. We have implemented a prototype and evaluated our approach on a set of local robustness analysis problem. The results show SURGEON not only can improve the analysis precision and is more effective than existing refinement techniques.

References

[1] M. Wu and L. Chen, "Image recognition based on deep learning," in *2015 Chinese Automation Congress (CAC)*. IEEE, 2015, pp. 542–546.

[2] M. Pak and S. Kim, "A review of deep learning in image recognition," in *2017 4th international conference on computer applications and information processing technology (CAIPT)*. IEEE, 2017, pp. 1–3.

[3] H. Zheng, J. Fu, T. Mei, and J. Luo, "Learning multi-attention convolutional neural network for fine-grained image recognition," in *Proceedings of the IEEE international conference on computer vision*, 2017, pp. 5209–5217.

[4] S. Ghosh and D. L. Reilly, "Credit card fraud detection with a neural-network," in *System Sciences, 1994. Proceedings of the Twenty-Seventh Hawaii International Conference on*, vol. 3. IEEE, 1994, pp. 621–630.

[5] R. Patidar, L. Sharma *et al.*, "Credit card fraud detection using neural network," *International Journal of Soft Computing and Engineering (IJSCE)*, vol. 1, no. 32-38, 2011.

[6] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *arXiv preprint arXiv:1409.0473*, 2014.

[7] K. Cho, B. Van Merriënboer, D. Bahdanau, and Y. Bengio, "On the properties of neural machine translation: Encoder-decoder approaches," *arXiv preprint arXiv:1409.1259*, 2014.

[8] M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang *et al.*, "End to end learning for self-driving cars," *arXiv preprint arXiv:1604.07316*, 2016.

[9] Q. Rao and J. Frtunikj, "Deep learning for self-driving cars: Chances and challenges," in *Proceedings of the 1st International Workshop on Software Engineering for AI in Autonomous Systems*, 2018, pp. 35–38.

[10] H. Kordylewski, D. Graupe, and K. Liu, "A novel large-memory neural network as an aid in medical diagnosis applications," *IEEE Transactions on Information Technology in Biomedicine*, vol. 5, no. 3, pp. 202–209, 2001.

[11] K. Kourou, T. P. Exarchos, K. P. Exarchos, M. V. Karamouzis, and D. I. Fotiadis, "Machine learning applications in cancer prognosis and prediction," *Computational and structural biotechnology journal*, vol. 13, pp. 8–17, 2015.

[12] G. Katz, C. Barrett, D. L. Dill, K. Julian, and M. J. Kochenderfer, "Reluplex: An efficient smt solver for verifying deep neural networks," in *International Conference on Computer Aided Verification*. Springer, 2017, pp. 97–117.

[13] R. Ehlers, "Formal verification of piece-wise linear feed-forward neural networks," in *International Symposium on Automated Technology for Verification and Analysis*. Springer, 2017, pp. 269–286.

[14] V. Tjeng, K. Xiao, and R. Tedrake, "Evaluating robustness of neural networks with mixed integer programming," *arXiv preprint arXiv:1711.07356*, 2017.

[15] T. Gehr, M. Mirman, D. Drachsler-Cohen, P. Tsankov, S. Chaudhuri, and M. Vechev, "Ai2: Safety and robustness certification of neural networks with abstract interpretation," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 3–18.

[16] G. Singh, T. Gehr, M. Mirman, M. Püschel, and M. Vechev, "Fast and effective robustness certification," in *Advances in Neural Information Processing Systems*, 2018, pp. 10 802–10 813.

[17] G. Singh, T. Gehr, M. Püschel, and M. Vechev, "An abstract domain for certifying neural networks," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, p. 41, 2019.

[18] S. Wang, K. Pei, J. Whitehouse, J. Yang, and S. Jana, "Formal security analysis of neural networks using sym-

bolic intervals," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, 2018, pp. 1599–1614.

[19] W. Shiqi, K. Pei, W. Justin, J. Yang, and S. Jana, "Efficient formal safety analysis of neural networks," in *32nd Conference on Neural Information Processing Systems (NIPS)*, Montreal, Canada, 2018.

[20] G. Anderson, S. Pailoor, I. Dillig, and S. Chaudhuri, "Optimization and abstraction: a synergistic approach for analyzing neural network robustness," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2019, pp. 731–744.

[21] J. Lu and M. P. Kumar, "Neural network branching for neural network verification," *arXiv preprint arXiv:1912.01329*, 2019.

[22] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot *et al.*, "Mastering the game of go with deep neural networks and tree search," *nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[23] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton *et al.*, "Mastering the game of go without human knowledge," *nature*, vol. 550, no. 7676, pp. 354–359, 2017.

[24] A. W. Senior, R. Evans, J. Jumper, J. Kirkpatrick, L. Sifre, T. Green, C. Qin, A. Žídek, A. W. Nelson, A. Bridgland *et al.*, "Improved protein structure prediction using potentials from deep learning," *Nature*, vol. 577, no. 7792, pp. 706–710, 2020.

[25] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus, "Intriguing properties of neural networks," *arXiv:1312.6199*, 2013.

[26] N. Carlini and D. Wagner, "Towards evaluating the robustness of neural networks," in *2017 ieee symposium on security and privacy (sp)*. IEEE, 2017, pp. 39–57.

[27] P. Cousot and R. Cousot, "Comparing the galois connection and widening/narrowing approaches to abstract interpretation," in *International Symposium on Programming Language Implementation and Logic Programming*. Springer, 1992, pp. 269–295.

[28] T. Miller, "Explanation in artificial intelligence: Insights from the social sciences," *Artificial intelligence*, vol. 267, pp. 1–38, 2019.

[29] MadryLab, "Mnist adversarial examples challenge," https://github.com/MadryLab/mnist_challenge.

[30] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu, "Towards deep learning models resistant to adversarial attacks," *arXiv preprint arXiv:1706.06083*, 2017.

[31] G. Singh, M. Püschel, and M. Vechev, "Making numerical program analysis fast," *ACM SIGPLAN Notices*, vol. 50, no. 6, pp. 303–313, 2015.

[32] ——, "Fast polyhedra abstract domain," in *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, 2017, pp. 46–59.

[33] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[34] T.-W. Weng, H. Zhang, H. Chen, Z. Song, C.-J. Hsieh, D. Boning, I. S. Dhillon, and L. Daniel, "Towards fast computation of certified robustness for relu networks," *arXiv preprint arXiv:1804.09699*, 2018.

[35] E. Wong and J. Z. Kolter, "Provable defenses against adversarial examples via the convex outer adversarial polytope," *arXiv preprint arXiv:1711.00851*, 2017.

[36] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement for symbolic model checking," *Journal of the ACM (JACM)*, vol. 50, no. 5, pp. 752–794, 2003.

[37] D. Beyer and S. Löwe, "Explicit-state software model checking based on cegar and interpolation," in *International Conference on Fundamental Approaches to Software Engineering*. Springer, 2013, pp. 146–162.

[38] M. Leucker, G. Markin, and M. R. Neuhäußer, "A new refinement strategy for cegar-based industrial model checking," in *Haifa Verification Conference*. Springer, 2015, pp. 155–170.

[39] S. Löwe, "Effective approaches to abstraction refinement for automatic software verification," 2017.

[40] K. L. McMillan, "Lazy abstraction with interpolants," in *International Conference on Computer Aided Verification*. Springer, 2006, pp. 123–136.

[41] E. M. Clarke, A. Gupta, and O. Strichman, "Sat-based counterexample-guided abstraction refinement," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 1113–1123, 2004.

[42] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement," in *International Conference on Computer Aided Verification*. Springer, 2000, pp. 154–169.

[43] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, "Lazy abstraction," in *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2002, pp. 58–70.

[44] T. Ball and S. K. Rajamani, "Automatically validating temporal safety properties of interfaces," in *International SPIN Workshop on Model Checking of Software*. Springer, 2001, pp. 102–122.

[45] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar, "The software model checker b last," *International Journal on Software Tools for Technology Transfer*, vol. 9, no. 5-6, pp. 505–525, 2007.

[46] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan, "Abstractions from proofs," *ACM SIGPLAN Notices*, vol. 39, no. 1, pp. 232–244, 2004.

[47] J. Li, J. Sun, L. Li, Q. L. Le, and S.-W. Lin, "Automatic loop-invariant generation and refinement through selective sampling," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 2017, pp. 782–792.