

# Reliability Assurance for Deep Neural Network Architectures Against Numerical Defects

Linyi Li\* Yuhao Zhang† Luyao Ren‡§ Yingfei Xiong‡§ Tao Xie‡§¶

\*Department of Computer Science, University of Illinois Urbana-Champaign, [linyi2@illinois.edu](mailto:linyi2@illinois.edu)

†Department of Computer Sciences, University of Wisconsin-Madison, [yuhao.zhang@wisc.edu](mailto:yuhao.zhang@wisc.edu)

‡School of Computer Science, Peking University, [rly@pku.edu.cn](mailto:rly@pku.edu.cn), [xiongyf@pku.edu.cn](mailto:xiongyf@pku.edu.cn), [taoxie@pku.edu.cn](mailto:taoxie@pku.edu.cn)

§Key Laboratory of High Confidence Software Technologies, Ministry of Education (Peking University)

**Abstract**—With the widespread deployment of deep neural networks (DNNs), ensuring the reliability of DNN-based systems is of great importance. Serious reliability issues such as system failures can be caused by numerical defects, one of the most frequent defects in DNNs. To assure high reliability against numerical defects, in this paper, we propose the RANUM approach including novel techniques for three reliability assurance tasks: detection of potential numerical defects, confirmation of potential-defect feasibility, and suggestion of defect fixes. To the best of our knowledge, RANUM is the first approach that confirms potential-defect feasibility with failure-exhibiting tests and suggests fixes automatically. Extensive experiments on the benchmarks of 63 real-world DNN architectures show that RANUM outperforms state-of-the-art approaches across the three reliability assurance tasks. In addition, when the RANUM-generated fixes are compared with developers’ fixes on open-source projects, in 37 out of 40 cases, RANUM-generated fixes are equivalent to or even better than human fixes.

**Index Terms**—neural network, numerical defect, testing, fix

## I. INTRODUCTION

Deep Neural Networks (DNNs) are successfully deployed and show remarkable performance in many challenging applications, including facial recognition [22, 57], game playing [39], and code completion [21, 4]. To develop and deploy DNNs, one needs to attain a DNN architecture, which is usually encoded by program code as the example shown in Figure 2. First, for training, the user executes the program with the architecture on the given training/validation data, attains the model weights, and stores them in a weight file. The architecture along with the weights is named a model. Then, for inference, the user loads the weight file to CPU/GPU memory or AI chips, executes the same program with the given inference sample and weights as arguments, and gets the model prediction result as the program output. With the wide deployment of DNN models (resulted from training DNN architectures), reliability issues of DNN-based systems have become a serious concern, where malfunctioning DNN-based systems have led to serious consequences such as fatal traffic accidents [25].

To assure the reliability of DNN-based systems, it is highly critical to detect and fix numerical defects for two main reasons. First, numerical defects widely exist in DNN-based systems. For example, in the DeepStability database [14], over

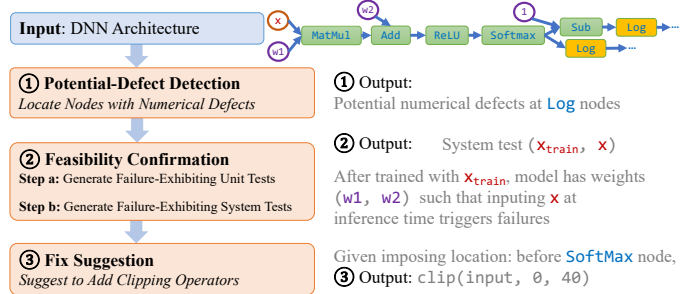


Fig. 1. Workflow for reliability assurance against numerical defects in DNN architectures. The left-hand side shows three tasks and the right-hand side shows corresponding examples. RANUM supports all the three tasks, and is the first automatic approach for system test generation and fix suggestion.

250 defects are identified in deep learning (DL) algorithms where over 60% of them are numerical defects. Moreover, since numerical defects exist at the architecture level, any model using the architecture naturally inherits these defects. Second, numerical defects can result in serious consequences. Once numerical defects (such as divide-by-zero) are exposed, the faulty DNN model will output NaN or INF instead of producing any meaningful prediction, resulting in numerical failures and system crashes [61, 59]. Thus, numerical defects hinder the application of DNNs in scenarios with high reliability and availability requirements such as threat monitoring in cybersecurity [34] and cloud system controlling [37, 13].

To address numerical defects in DNN architectures in an actionable manner [54], in this paper, we propose a workflow of reliability assurance (as shown in Figure 1), consisting of three tasks: potential-defect detection, feasibility confirmation, and fix suggestion, along with our proposed approach to support all these three tasks.

**Potential-Defect Detection.** In this task, we detect all potential numerical defects in a DNN architecture, with a focus on operators with numerical defects (in short as defective operators) that potentially exhibit inference-phase numerical failures for two main reasons, following the literature [62, 55]. First, these defective operators can be exposed after the model is deployed and thus are more devastating than those that potentially exhibit training-phase numerical failures [28, 62]. Second, a defective operator that potentially exhibits training-phase numerical failures can usually be triggered to exhibit inference-phase numerical failures, thus also being detected by

¶Corresponding author.

our task. For example, the type of training-phase NaN gradient failures is caused by an operator’s input that leads to invalid derivatives, and this input also triggers failures in the inference phase [55].

*Feasibility Confirmation.* In this task, we confirm the feasibility of these potential numerical defects by generating failure-exhibiting system tests. As shown in Figure 1, a system test is a tuple of training example<sup>1</sup>  $x_{\text{train}}$  and inference example  $x$  such that after the training example is used to train the architecture under consideration, applying the resulting model on the inference example exhibits a numerical failure.

*Fix Suggestion.* In this task, we fix a feasible numerical defect. To determine the fix form, we have inspected the developers’ fixes of the numerical defects collected by Zhang et al. [61] by looking at follow-up Stack Overflow posts or GitHub commits. Among the 13 numerical defects whose fixes can be located, 12 fixes can be viewed as explicitly or implicitly imposing interval preconditions on different locations, such as after inputs or weights are loaded and before defective operators are invoked. Thus, imposing an interval precondition, e.g., by clipping (i.e., chopping off the input parts that exceed the specified input range) the input for defective operator(s), is an effective and common strategy for fixing a numerical defect. Given a location (i.e., one related to an operator, input, or weight where users prefer to impose a fix), we suggest a fix for the numerical defect under consideration.

To support all the three tasks of the **reliability assurance** process against DNN **numerical** defects, we propose the **RANUM** approach in this paper.

For task ① and task ②a, which are already supported by two existing tools (DEBAR [62] and GRIST [55]), RANUM introduces novel extensions and optimizations that substantially improve the effectiveness and efficiency. (1) DEBAR [62] is the state-of-the-art tool for potential-defect detection; however, DEBAR can handle only static computational graphs and does not support widely used dynamic graphs in PyTorch programs [29]. RANUM supports dynamic graphs thanks to our novel technique of *backward fine-grained node labeling*. (2) GRIST [55] is the state-of-the-art tool for generating failure-exhibiting unit tests to confirm potential-defect feasibility; however, GRIST conducts gradient back-propagation by using the original inference input and weights as the starting point. Recent studies [24, 8] on DNN adversarial attacks suggest that using a randomized input as the starting point leads to stronger attacks than using the original input. Taking this observation, we combine gradient back-propagation with random initialization in RANUM.

For task ② and task ③, which are not supported by any existing tool, RANUM is the first automatic approach for them.

For **feasibility confirmation**, RANUM is the **first** approach that generates failure-exhibiting **system** tests that contain train-

ing examples. Doing so is a major step further from the existing GRIST tool, which generates failure-exhibiting unit tests ignoring the practicality of generated model weights. Given that in practice model weights are determined by training examples, we propose the technique of *two-step generation* for this task. First, we generate a failure-exhibiting unit test. Second, we generate a training example that leads to the model weights in the unit test when used for training. For the second step, we extend the deep-leakage-from-gradient (DLG) attack [64] by incorporating the straight-through gradient estimator [3].

For **fix suggestion**, RANUM is the **first** automatic approach. RANUM is based on the novel technique of *abstraction optimization*. We observe that a defect fix in practice is typically imposing interval clipping on some operators such that each later-executed operator (including those defective ones) can never exhibit numerical failures. Therefore, we propose the novel technique of abstraction optimization to “deviate away” the input range of a defective operator from the invalid range, falling in which can cause numerical failures.

For RANUM, we implement a tool<sup>2</sup> and evaluate it on the benchmarks [55] of 63 real-world DNN architectures containing 79 real numerical defects; these benchmarks are the largest benchmarks of DNN numerical defects to the best of our knowledge. The evaluation results show that RANUM is both effective and efficient in all the three tasks for DNN reliability assurance. (1) For potential-defect detection, RANUM detects >60% more true defects than the state-of-the-art DEBAR approach. (2) For feasibility confirmation, RANUM generates failure-exhibiting unit tests to confirm potential numerical defects in the benchmarks with 100% success rate; in contrast, with the much higher time cost (17.32X), the state-of-the-art GRIST approach generates unit tests to confirm defects with 96.96% success rate. More importantly, for the first time, RANUM generates failure-exhibiting system tests that confirm defects (with 92.78% success rate). (3) For fix suggestion, RANUM proposes fix suggestions for numerical defects with 100% success rate. In addition, when the RANUM-generated fixes are compared with developers’ fixes on open-source projects, in 37 out of 40 cases, RANUM-generated fixes are equivalent to or even better than human fixes.

This paper makes the following main contributions:

- We formulate the reliability assurance problem for DNN architectures against numerical defects and elaborate on three important tasks for this problem.
- We propose RANUM—the first automatic approach that solves all these three tasks. RANUM includes three novel techniques (backward fine-grained node labeling, two-step test generation, and abstraction optimization) and solves system test generation and fix suggestion for the first time.
- We implement RANUM and apply it on 63 real-world DNN architectures, showing the high effectiveness and efficiency of RANUM compared to both the state-of-the-art approaches and developers’ fixes.

<sup>1</sup>In real settings, multiple training examples are used to train an architecture, but generating a single training example to exhibit failures (targeted by our work) is desirable for ease of debugging while being more challenging than generating multiple training examples to exhibit failures.

<sup>2</sup>Open source at <https://github.com/llylly/RANUM>.

```

1 input_data = tf.placeholder("float", [1, n_features], name='x-input')
2 input_labels = tf.placeholder("float", [1, n_classes], name='y-input')
3 self.W_ = tf.Variable(tf.zeros([n_features, n_classes]),
4                       name='weights')
5 self.b_ = tf.Variable(tf.zeros([n_classes]),
6                       name='biases')
7 model_output = tf.nn.softmax(tf.matmul(input_data, self.W_) +
8                              self.b_)
9 cost = -tf.reduce_mean(input_labels * tf.log(model_output) +
10                       (1 - input_labels) * tf.log(1 - model_output),
11                       name='cost')
12 self.obj_function = tf.reduce_min(tf.abs(model_output),
13                                  name='obj_function')

```

Fig. 2. A DL program snippet that defines a linear regression model from benchmarks of real-world numerical defects (Case 2a in [55]).

## II. BACKGROUND AND APPROACH OVERVIEW

In this section, we introduce the background of DNN numerical defects and failures, and then give an overview of the RANUM approach with a running example.

### A. Background

DL developers define the DNN architecture with code using modern DL libraries such as PyTorch [29] and TensorFlow [1]. The DNN architecture can be expressed by a computational graph. Figures 2 and 3 depict a real-world example. Specifically, the DNN architecture in a DL program can be automatically converted to an ONNX-format computational graph [44].

The computational graph can be viewed as a Directed Acyclic Graph (DAG):  $\mathcal{G} = \langle \mathcal{V}, \mathcal{E} \rangle$ , where  $\mathcal{V}$  and  $\mathcal{E}$  are sets of nodes and edges, respectively. We call nodes with zero in-degree as *initial nodes*, which correspond to input, weight, or constant nodes. Initial nodes provide concrete data for the DNN models resulted from training the DNN architecture. The data from each node is formatted as a tensor, i.e., a multidimensional array, with a specified data type and array shape annotated alongside the node definition. We call nodes with positive in-degree as *internal nodes*, which correspond to concrete operators, such as matrix multiplication (`MatMul`) and addition (`Add`). During model training, the model weights, i.e., data from weight nodes, are generated by the training algorithm. Then, in the deployment phase (i.e., model inference), with these trained weights and a user-specified input named inference example, the output of each operator is computed in the topological order. The output of some specific node is used as the prediction result.

We let  $x$  and  $w$  denote the concatenation of data from all input nodes and data from all weight nodes, respectively.<sup>3</sup> For example, in Figure 3,  $x$  concatenates data from nodes 1 and 11; and  $w$  concatenates data from nodes 2 and 4. Given specific  $x$  and  $w$ , the input and output for each node are deterministic.<sup>4</sup> We use  $f_n^{\text{in}}(x; w)$  and  $f_n^{\text{out}}(x; w)$  to express input and output data of node  $n$ , respectively, given  $x$  and  $w$ . **Numerical Defects in DNN Architecture.** We focus on inference-phase numerical defects. These defects lead to numerical failures when specific operators receive inputs within invalid ranges so that the operators output NaN or INF.

<sup>3</sup>A bolded alphabet stands for a vector or tensor throughout the paper.

<sup>4</sup>An architecture may contain stochastic nodes. We view these nodes as nodes with randomly sampled data, so the architecture itself is deterministic.

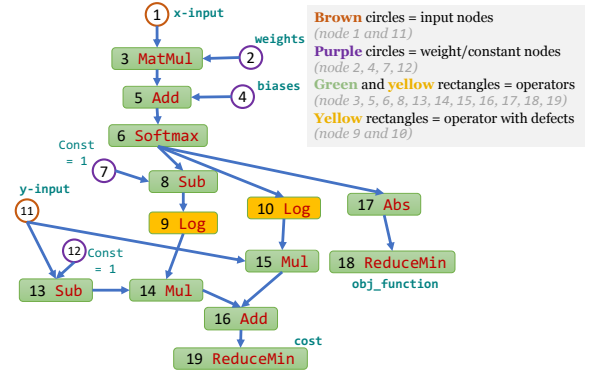


Fig. 3. Computational graph encoded by the snippet in Figure 2.

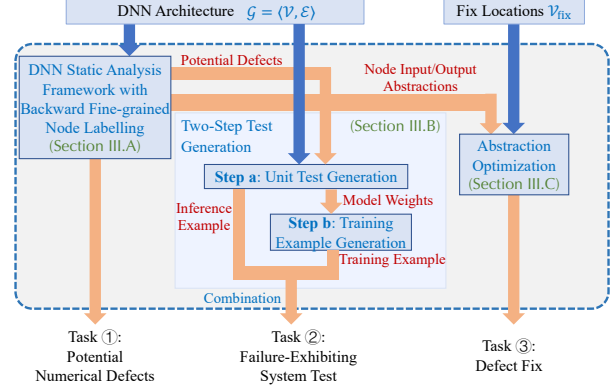


Fig. 4. Overview of the RANUM approach. The output of RANUM indicates confirmation and manifestation of numerical defects (that can be feasibly exposed at the system level) for a given DNN architecture and effective fixes for the architecture's confirmed defects.

**Definition 1.** For the given computational graph  $\mathcal{G} = \langle \mathcal{V}, \mathcal{E} \rangle$ , if there is a node  $n_0 \in \mathcal{V}$ , such that there exists a valid input and valid weights that can let the input of node  $n_0$  fall within the invalid range, we say there is a **numerical defect** at node  $n_0$ .

Formally,  $\exists x_0 \in \mathcal{X}_{\text{valid}}, w_0 \in \mathcal{W}_{\text{valid}}, f_{n_0}^{\text{in}}(x_0; w_0) \in \mathcal{I}_{n_0, \text{invalid}} \implies \exists$  numerical defect at node  $n_0$ .

In the definition,  $\mathcal{X}_{\text{valid}}$  and  $\mathcal{W}_{\text{valid}}$  are valid input range and weight range, respectively, which are clear given the deployed scenario. For example, ImageNet Resnet50 models have valid input range  $\mathcal{X}_{\text{valid}} = [0, 1]^{3 \times 224 \times 224}$  since image pixel intensities are within  $[0, 1]$ , and valid weight range  $\mathcal{W}_{\text{valid}} = [-1, 1]^p$  where  $p$  is the number of parameters since weights of well-trained Resnet50 models are typically within  $[-1, 1]$ . The invalid range  $\mathcal{I}_{n_0, \text{invalid}}$  is determined by  $n_0$ 's operator type with detailed definitions in Suppl. B. For example, for the `Log` operator, the invalid range  $\mathcal{I}_{n_0, \text{invalid}} = (-\infty, U_{\text{min}})$  where  $U_{\text{min}}$  is the smallest positive number of a tensor's data type.

### B. Approach Overview

In Figure 4, we show the overview structure of the RANUM approach. RANUM takes a DNN architecture as the input. Note that although RANUM is mainly designed and illustrated for a DNN architecture, the RANUM approach can also be directly applied to general neural network architectures since they can also be expressed by computational graphs. First, the DNN static analysis framework (task ① in Figure 1) in

RANUM detects all potential numerical defects in the architecture. Second, the two-step test generation component (task ② in Figure 1), including unit test generation and training example generation, confirms the feasibility of these potential numerical defects. Third, the abstraction optimization component (task ③ in Figure 1) takes the input/output abstractions produced by the DNN static analysis framework along with the user-specified fix locations, and produces preconditions to fix the confirmed defects.

We next go through the whole process in detail taking the DNN architecture shown in Figure 3 as a running example.

**Task ①: Potential-Defect Detection via Static Analysis.**

The DNN static analysis framework within RANUM first computes the numerical intervals of possible inputs and outputs for all nodes within the given DNN architecture, and then flags any nodes whose input intervals overlap with their invalid ranges as nodes with potential numerical defects.

In Figure 3, suppose that the user-specified input  $x$ -input (node 1) is within (elementwise, same below) range  $[(-10, -10)^\top, (10, 10)^\top]$ ; weights (node 2) are within range  $\begin{bmatrix} -10 & -10 \\ -10 & -10 \end{bmatrix}$ ,  $\begin{bmatrix} 10 & 10 \\ 10 & 10 \end{bmatrix}$ ; and biases (node 4) are within range  $[(-10, -10)^\top, (10, 10)^\top]$ . Our DNN static analysis framework computes these interval abstractions for node inputs:

- 1) Node 5 (after MatMul):  $[(-200, -200)^\top, (200, 200)^\top]$ ;
- 2) Node 6 (after Add):  $[(-210, -210)^\top, (210, 210)^\top]$ ;
- 3) Node 8 (after Softmax in float32):  $[(0, 0)^\top, (1, 1)^\top]$ ;
- 4) Node 9 (after Sub of  $[1, 1]$  and node 8), 10:  $[(0, 0)^\top, (1, 1)^\top]$ .

Since nodes 9 and 10 use the `Log` operator whose invalid input range  $(-\infty, U_{\min})$  overlaps with their input range  $[(0, 0)^\top, (1, 1)^\top]$ , we flag nodes 9 and 10 as potential numerical defects.

This static analysis process follows the state-of-the-art DEBAR tool [62]. However, we extend DEBAR with a novel technique named backward fine-grained node labeling. This technique detects all nodes that require fine-grained abstractions, e.g., nodes that determine the control flow in a dynamic graph. For these nodes, we apply interval abstractions with the finest granularity to reduce control flow ambiguity. For other nodes, we let some neighboring elements share the same interval abstraction to improve efficiency while preserving tightness. As a result, the static analysis in RANUM has high efficiency and supports much more DNN operators including dynamic control-flow operators like `Loop` than DEBAR does.

**Task ②: Feasibility Confirmation via Two-Step Test Generation.**

Given nodes that contain potential numerical defects (nodes 9 and 10 in our example), we generate failure-exhibiting system tests to confirm their feasibility. A failure-exhibiting system test is a tuple  $\langle \mathbf{x}_{\text{train}}, \mathbf{x}_{\text{infer}} \rangle$ , such that after training the architecture with the training example  $\mathbf{x}_{\text{train}}$ ,<sup>5</sup> with the trained model weights  $\mathbf{w}_{\text{infer}}$ , the inference input

<sup>5</sup>In particular, if our generation technique outputs  $\mathbf{x}_{\text{train}}$ , the numerical failure can be triggered if the training dataset contains *only*  $\mathbf{x}_{\text{train}}$  or *only* multiple copies of  $\mathbf{x}_{\text{train}}$  and the inference-time input is  $\mathbf{x}_{\text{infer}}$ . Our technique can also be applied for generating a batch of training examples by packing the batch as a single example:  $\mathbf{x}_{\text{train}} = (\mathbf{x}_{\text{train}1}, \mathbf{x}_{\text{train}2}, \dots, \mathbf{x}_{\text{train}B})$ .

$\mathbf{x}_{\text{infer}}$  triggers a numerical failure. The name “system test” is inspired by traditional software testing, where we test the method sequence ( $m = \text{train}(\mathbf{x}_{\text{train}}); m.\text{infer}(\mathbf{x}_{\text{infer}})$ ). In contrast, GRIST [55] generates model weights  $\mathbf{w}_{\text{infer}}$  along with inference input  $\mathbf{x}_{\text{infer}}$  that tests only the inference method  $m.\text{infer}()$ , and the weights may be infeasible from training. Hence, we view the GRIST-generated tuple  $\langle \mathbf{w}_{\text{infer}}, \mathbf{x}_{\text{infer}} \rangle$  as a “unit test”.

We propose a two-step test generation technique to generate failure-exhibiting system tests.

**Step a: Generate failure-exhibiting unit test  $\langle \mathbf{w}_{\text{infer}}, \mathbf{x}_{\text{infer}} \rangle$ .**

The state-of-the-art GRIST tool supports this step. However, GRIST solely relies on gradient back-propagation, which is relatively inefficient. In RANUM, we augment GRIST by combining its gradient back-propagation with random initialization inspired by recent research on DNN adversarial attacks [8, 24]. As a result, RANUM achieves 17.32X speedup with 100% success rate. Back to the running example in Figure 3, RANUM can generate  $\begin{bmatrix} 5 & -5 \\ -5 & 5 \end{bmatrix}$  for node 2 and  $(0.9, -0.9)^\top$  for node 4 as model weights  $\mathbf{w}_{\text{infer}}$ ; and  $(10, -10)^\top$  for node 1 and  $(1, 0)^\top$  for node 11 as the inference input  $\mathbf{x}_{\text{infer}}$ . Such  $\mathbf{w}_{\text{infer}}$  and  $\mathbf{x}_{\text{infer}}$  induce input  $(0, 1)^\top$  and  $(1, 0)^\top$  for nodes 9 and 10, respectively. Since both nodes 9 and 10 use the `log` operator and `log 0` is undefined, both nodes 9 and 10 trigger numerical failures.

**Step b: Generate training example  $\mathbf{x}_{\text{train}}$  that achieves model weights  $\mathbf{w}_{\text{infer}}$ .**

To the best of our knowledge, there is no automatic approach for this task yet. RANUM provides support for this task based on our extension of DLG attack [64]. The DLG attack is originally designed for recovering the training data from training-phase gradient leakage. Here, we figure out the required training gradients to trigger the numerical failure at the inference phase and then leverage the DLG attack to generate  $\mathbf{x}_{\text{train}}$  that leads to such training gradients. Specifically, many DNN architectures contain operators (such as `ReLU`) on which DLG attack is hard to operate [36]. We combine straight-through estimator [3] to provide proxy gradients and bypass this barrier. Back to the running example in Figure 3, supposing that the initial weights are  $\begin{bmatrix} -0.1 & 0.1 \\ 0.1 & -0.1 \end{bmatrix}$  for node 2 and  $(0, 0)^\top$  for node 4, RANUM can generate training example  $\mathbf{x}_{\text{train}}$  composed of  $(5.635, -5.635)^\top$  for node 1 and  $(1, 0)^\top$  for node 11, such that one-step training with learning rate 1 on this example leads to  $\mathbf{w}_{\text{infer}}$ . Combining  $\mathbf{x}_{\text{train}}$  from this step with  $\mathbf{x}_{\text{infer}}$  from *step a*, we obtain a failure-exhibiting system test that confirms the feasibility of potential defects in nodes 9 and 10.

**Task ③: Fix Suggestion via Abstract Optimization.**

In this task, we suggest fixes for the confirmed numerical defects. RANUM is the first approach for this task to our knowledge. The user may prefer different fix locations, which correspond to a user-specified set of nodes  $\mathcal{V}_{\text{fix}} \subseteq \mathcal{V}$  to impose the fix. For example, if the fix method is clipping the inference input,  $\mathcal{V}_{\text{fix}}$  are input nodes (e.g., nodes 1, 11 in Figure 3); if the fix method is clipping the model weights during training,  $\mathcal{V}_{\text{fix}}$  are weight nodes (e.g., nodes 2, 4 in Figure 3); if the fix

method is clipping before the defective operator,  $\mathcal{V}_{\text{fix}}$  are nodes with numerical defects (e.g., nodes 9, 10 in Figure 3).

According to the empirical study of developers’ fixes in Section I, 12 out of 13 defects are fixed by imposing interval preconditions for clipping the inputs of  $\mathcal{V}_{\text{fix}}$ . Hence, we suggest interval precondition, which is interval constraint  $l_n \leq f_n^{\text{in}}(\mathbf{x}; \mathbf{w}) \leq u_n$  for nodes  $n \in \mathcal{V}_{\text{fix}}$ , as the defect fix in this paper. A fix should satisfy that, when these constraints  $\bigwedge_{n \in \mathcal{V}_{\text{fix}}} (l_n \leq f_n^{\text{in}}(\mathbf{x}; \mathbf{w}) \leq u_n)$  are imposed, the input of any node in the computational graph should always be valid, i.e.,  $f_{n_0}^{\text{in}}(\mathbf{x}; \mathbf{w}) \notin \mathcal{I}_{n_0, \text{invalid}}, \forall n_0 \in \mathcal{V}$ .

In RANUM, we formulate the fix suggestion task as a constrained optimization problem, taking the endpoints of interval abstractions for nodes in  $\mathcal{V}_{\text{fix}}$  as optimizable variables. We then propose the novel technique of abstraction optimization to solve this constrained optimization problem. Back to the Figure 3 example, if users plan to impose a fix on inference input, RANUM can suggest the fix  $-1 \leq \text{x-input} \leq 1$ ; if users plan to impose a fix on nodes with numerical defects, RANUM can suggest the fix  $10^{-38} \leq \text{node 9 \& node 10.input} \leq +\infty$ .

### III. THE RANUM APPROACH

In this section, we introduce the three novel techniques in RANUM: backward fine-grained node labeling in Section III-A; two-step test generation in Section III-B; and abstraction optimization in Section III-C.

#### A. DNN Static Analysis Framework with Backward Fine-Grained Node Labeling for Potential-Defect Detection

RANUM contains a static analysis framework to enable potential-defect detection and support downstream tasks as shown in Figure 4. Given a DNN architecture and valid ranges for input and weight nodes, the static analysis framework computes interval abstractions for possible inputs and outputs of each node. As a result, we can check whether an overlap exists between the interval abstraction and invalid input ranges for all nodes in the graph to detect potential numerical defects. Then, the defective nodes are fed into the two-step test generation component to confirm the feasibility of potential defects; and the differentiable abstractions are fed into the abstract optimization component to produce fixes.

Formally, for given valid ranges of inference input and model weights, namely  $\mathcal{X}$  and  $\mathcal{W}$ , for each node  $n \in \mathcal{V}$ , our framework computes *sound* input interval abstraction  $[l_n, u_n] := \{\mathbf{x} : l_n \leq \mathbf{x} \leq u_n\}$  such that  $[l_n, u_n]$  always captures all possible inputs of the node:  $[l_n, u_n] \supseteq \{f_n^{\text{in}}(\mathbf{x}, \mathbf{w}) : \mathbf{x} \in \mathcal{X}, \mathbf{w} \in \mathcal{W}\}$ . We also compute output interval abstractions similarly.

Compared with traditional analysis tools for numerical software [10, 40], RANUM’s static analysis framework designs abstractions for DNN primitives operating on multi-dimensional tensors that are not supported by traditional tools. Compared with the state-of-the-art DEBAR tool [62], RANUM uses the same abstraction domain (interval domain with tensor partitioning), but incorporates a novel tech-

nique (backward fine-grained node labeling) to improve abstraction precision and support a wider range of DNN architectures.

**Abstract Domain: Interval with Tensor Partitioning.** Following DEBAR’s design, we use the interval with tensor partitioning [62] as the abstraction domain. This abstraction domain partitions a tensor into multiple subblocks and shares the interval abstractions at the block level instead of imposing abstractions at the element level. Therefore, we can compute the abstraction of a smaller size than the original tensor to improve efficiency.

**Our Technique: Backward Fine-Grained Node Labeling.** The interval domain with tensor partitioning provides a degree of freedom in terms of the partition granularity, i.e., we can choose the subblock size for each node’s abstraction. When the finest granularity, i.e., elementwise abstraction, is chosen, the abstraction interval is the most concrete. When the coarsest granularity (i.e., one scalar to summarize the node tensor) is chosen, the abstraction saves the most space and computational cost but loses much precision.

*Example.* Suppose that the possible input range of a node is  $([-1, 0], [0, 1], [1, 2], [-1, 0])$ , where each interval  $[l, u]$  specifies the range of corresponding elements in the four-dimensional vector. If we choose the finest granularity, we use  $[l_n, u_n] = [(-1, 0, 1, -1), (0, 1, 2, 0)]$  as the input range abstraction. If we choose the coarsest granularity, we use  $[l_n, u_n] = [-1, 2]$  as the abstraction where the same interval is shared for all elements. As we can see, finer granularity provides tighter abstraction at the expense of larger computational and space costs.

In DEBAR, the coarsest granularity is used by default for most operators. However, we find that using the finest instead of the coarsest granularity for some nodes is more beneficial for overall abstraction preciseness. For example, the control-flow operators (e.g., `Loop`) benefit from concrete execution to determine the exact control flow in the dynamic graph, and the indexing operators (e.g., `slice`) and shaping operators (e.g., `Reshape`) benefit from explicit indexers and shapes to precisely infer the output range. Hence, we propose to use the finest granularity for some nodes (namely fine-grained requiring operators) while the coarsest granularity for other nodes during static analysis.

To benefit from the finest granularity abstraction for required nodes, typically, all of their preceding nodes also need the finest granularity. Otherwise, the over-approximated intervals from preceding nodes will be propagated to the required nodes, making the finest abstraction for the required nodes useless. To solve this problem, in RANUM, we back-propagate “fine-grained” labels from these fine-grained requiring nodes to initial nodes by topologically sorting the graph with *inverted* edges, and then apply the finest granularity abstractions on all labeled nodes. In practice, we find that this strategy eliminates the control-flow ambiguity and indexing ambiguity with little

loss of efficiency<sup>6</sup>. As a result, RANUM supports all dynamic graphs (which are not supported by DEBAR) that comprise 39.2% of the benchmarks proposed by Yan et al. [55].

Furthermore, when preceding nodes use finer-grain abstraction granularity, the subsequent nodes should preserve such fine granularity to preserve the analysis preciseness. Principally, the choice of abstraction granularity should satisfy both tightness (bearing no precision loss compared to element-wise interval abstraction) and minimality (using the minimum number of partitions for high efficiency). To realize these principles, we dynamically determine a node’s abstraction granularity based on the granularity of preceding nodes. The abstraction design for some operators is non-trivial. Omitted details (formulation, illustration, and proofs) about the static analysis framework are in Suppl. C.

In summary, the whole static analysis process consists of three steps. (1) Determine the tensor partition granularity of all initial nodes by our technique of backward fine-grained node labeling. (2) Sort all nodes in the graph in the topological order. (3) Apply corresponding abstraction computation algorithms for each node based on the preceding node’s abstractions. The key insight behind the design of our static analysis framework is the strategic granularity selection for tensor abstraction, maintaining both high efficiency (by selecting the coarse granularity for data-intensive nodes) and high precision (by selecting the fine granularity for some critical nodes, such as nodes with control-flow, indexing, and shaping operators).

### B. Two-Step Test Generation for Feasibility Confirmation

RANUM generates failure-exhibiting system tests for the given DNN to confirm the feasibility of potential numerical defects. Here, we take the DNN architecture as the input. From the static analysis framework, we obtain a list of nodes that have potential numerical defects. For each node  $n_0$  within the list, we apply our technique of two-step test generation to produce a failure-exhibiting system test  $t_{\text{sys}} = \langle \mathbf{x}_{\text{train}}, \mathbf{x}_{\text{infer}} \rangle$  as the output. According to Section II-B, the test should satisfy that after the architecture is trained with  $\mathbf{x}_{\text{train}}$ , entering  $\mathbf{x}_{\text{infer}}$  in the inference phase results in a numerical failure.

We propose the novel technique of two-step test generation: first, generate failure-exhibiting unit test  $\langle \mathbf{w}_{\text{infer}}, \mathbf{x}_{\text{infer}} \rangle$ ; then, generate training example  $\mathbf{x}_{\text{train}}$  that leads model weights to be close to  $\mathbf{w}_{\text{infer}}$  after training.

**Step a: Unit Test Generation.** As sketched in Section II-B, we strengthen the state-of-the-art unit test generation approach, GRIST [55], by combining it with random initialization to complete this step. Specifically, GRIST leverages the gradients of the defective node’s input with respect to the inference input and weights to iteratively update the inference input and weights to generate failure-exhibiting unit tests. However,

<sup>6</sup>Theoretically, using the finest granularity for tensor partitioning cannot fully eliminate the ambiguity, since interval abstraction is intrinsically an over-approximation. Nevertheless, in our evaluation (Section IV), we find that this technique eliminates control-flow and indexing ambiguities on all 63 programs in the benchmarks.

GRIST always conducts updates from the existing inference input and weights, suffering from local minima problem [24]. Instead, motivated by DNN adversarial attack literature [24, 46], a sufficient number of random starts help find global minima effectively. Hence, in RANUM, we first conduct uniform sampling 100 times for both the inference input and weights to trigger the numerical failure. If no failure is triggered, we use the sample that induces the smallest loss as the start point for gradient optimization. As Section IV-A shows, this strategy substantially boosts the efficiency, achieving 17.32X speedup.

**Step b: Training Example Generation.** For this step, RANUM takes the following inputs: (1) the DNN architecture, (2) the failure-exhibiting unit test  $t_{\text{unit}} = \langle \mathbf{w}_{\text{infer}}, \mathbf{x}_{\text{infer}} \rangle$ , and (3) the randomly initialized weights  $\mathbf{w}_0$ . Our goal is to generate a legal training example  $\mathbf{x}_{\text{train}}$ , such that the model trained with  $\mathbf{x}_{\text{train}}$  will contain weights close to  $\mathbf{w}_{\text{infer}}$ .

DNNs are typically trained with gradient-descent-based algorithms such as stochastic gradient descent (SGD). In SGD, in each step  $t$ , we sample a mini-batch of samples from the training dataset to compute their gradients on model weights and use these gradients to update the weights. We focus on one-step SGD training with a single training example, since generating a single one-step training example to exhibit a failure is more desirable for debugging because, in one-step training, the model weights are updated strictly following the direction of the gradients. Therefore, developers can inspect inappropriate weights, easily trace back to nodes with inappropriate gradients, and then fix these nodes. In contrast, in multi-step training, from inappropriate weights, developers cannot trace back to inappropriate gradients because weights are updated iteratively and interactions between gradients and weights are complex (even theoretically intractable [19]).

In this one-step training case, after training, the model weights  $\mathbf{w}_{\text{infer}}$  satisfy

$$\mathbf{w}_{\text{infer}} = \mathbf{w}_0 - \gamma \nabla_{\mathbf{w}} \mathcal{L}(\mathbf{x}_{\text{train}}; \mathbf{w}_0), \quad (1)$$

where  $\gamma \in \mathbb{R}_+$  is a predefined learning rate, and  $\mathcal{L}$  is the predefined loss function in the DNN architecture. Hence, our goal becomes finding  $\mathbf{x}_{\text{train}}$  that satisfies

$$\nabla_{\mathbf{w}} \mathcal{L}(\mathbf{x}_{\text{train}}; \mathbf{w}_0) = (\mathbf{w}_0 - \mathbf{w}_{\text{infer}}) / \gamma. \quad (2)$$

The DLG attack [64] is a technique for generating input data that induce specific weight gradients. The attack is originally designed for recovering training samples from monitored gradient updates. Since the right-hand side (RHS) of Equation (2) is known, our goal here is also to generate input example  $\mathbf{x}_{\text{train}}$  that induces specific weight gradients. Therefore, we leverage the DLG attack to generate training example  $\mathbf{x}_{\text{train}}$ .

**Extending DLG Attack with Straight-Through Estimator.** Directly using DLG attack suffers from an optimization challenge in our scenario. Specifically, in DLG attack, suppose that the target weight gradients are  $\Delta \mathbf{w}_{\text{targ}}$ , we use gradient descent over the squared error  $\|\nabla_{\mathbf{w}} \mathcal{L}(\mathbf{x}; \mathbf{w}_0) - \Delta \mathbf{w}_{\text{targ}}\|_2^2$  to generate  $\mathbf{x}$ . In this process, we need meaningful gradient information of this squared error loss to perform the optimization. However, the gradient of this loss involves second-order derivatives of  $\mathcal{L}(\mathbf{x}; \mathbf{w}_0)$ , which could be zero. For

example, DNNs with ReLU as activation function are piecewise linear and have zero second-order derivatives almost everywhere [36]. This optimization challenge is partly addressed in DLG attack by replacing ReLU with Sigmoid, but it changes the DNN architecture (i.e., the system under test) and hence is unsuitable.

We leverage the straight-through estimator to mitigate the optimization challenge. Specifically, for a certain operator, such as ReLU, we do not change its forward computation but change its backward gradient computation to provide second-order derivatives within the DLG attack process. For example, for ReLU, in backward computation we use the gradient of Softplus function, namely  $1 - \frac{1}{1 + \exp(\mathbf{x})}$ , because Softplus is an approximation of ReLU [7] with non-zero second-order derivatives. Note that we modify the computed gradients only within the DLG attack process. After such  $\mathbf{x}_{\text{train}}$  is generated by the attack, we evaluate whether it triggers a numerical failure using the original architecture and gradients in Equation (1).

Suppl. E lists hyperparameters used by our implementation.

### C. Abstraction Optimization for Fix Suggestion

In this task, we aim to generate the precondition fix given imposing locations. The inputs are the DNN architecture, the node  $n_0$  with numerical defects, and a node set  $\mathcal{V}_{\text{fix}}$  to impose the fix. We would like to generate interval preconditions for  $\mathcal{V}_{\text{fix}}$  node inputs so that after these preconditions are imposed, the defect on  $n_0$  is fixed.

Formally, our task is to find  $\langle l_n, u_n \rangle$  for each  $n \in \mathcal{V}_{\text{fix}}$  ( $l_n$  and  $u_n$  are scalars so the same interval bound applied to all elements of  $n$ 's tensor), such that for any  $\mathbf{x}, \mathbf{w}$  satisfying  $f_n^{\text{in}}(\mathbf{x}; \mathbf{w}) \in [l_n, u_n]$ ,  $\forall n \in \mathcal{V}_{\text{fix}}$ , for the defective node  $n_0$ , we have  $f_{n_0}^{\text{in}}(\mathbf{x}; \mathbf{w}) \notin \mathcal{I}_{n_0, \text{invalid}}$ , where the full list of invalid input ranges  $\mathcal{I}_{n_0, \text{invalid}}$  is in Suppl. B. There is an infinite number of possible  $\langle l_n, u_n \rangle$  interval candidates since  $l_n$  and  $u_n$  are floating numbers. Hence, we need an effective technique to find a valid solution from the exceedingly large search space that incurs a relatively small model utility loss. To achieve so, we formulate a surrogate optimization problem for this task.

$$\underset{l_n, u_n : n \in \mathcal{V}_{\text{fix}}}{\text{maximize}} \quad s \quad \text{s.t.} \quad u_n \geq l_n + s(u_n^{\text{valid}} - l_n^{\text{valid}}), \forall n \in \mathcal{V}_{\text{fix}}, \quad (3)$$

$$l_n^{\text{valid}} \leq l_n \leq u_n \leq u_n^{\text{valid}}, \forall n \in \mathcal{V}_{\text{fix}}, \quad (4)$$

$$\mathcal{L}_{n_0}^{\text{precond}}(\{l_n, u_n\}_{n \in \mathcal{V}_{\text{fix}}}) < 0. \quad (5)$$

Here,  $l_n^{\text{valid}}$  and  $u_n^{\text{valid}}$  are the valid ranges (of the node's input  $n$ ), which are fixed and determined by the valid ranges of input and weights.  $\mathcal{L}_{n_0}^{\text{precond}}$  is the node-specific precondition generation loss that is the distance between the furthest endpoint of defective node  $n_0$ 's interval abstraction and  $n_0$ 's valid input range. Hence, when  $\mathcal{L}_{n_0}^{\text{precond}}(\{l_n, u_n\}_{n \in \mathcal{V}_{\text{fix}}})$  becomes negative, the solution  $\{l_n, u_n\}_{n \in \mathcal{V}_{\text{fix}}}$  is a valid precondition. The optimization variables are the precondition interval endpoints  $l_n$  and  $u_n$  and the objective is the relative span of these intervals. The larger the span is, the looser the precondition constraints are, and the less hurt they are for the model's utility. Equation (3) enforces the interval span requirement. Equation (4) assures that the precondition interval is in the valid range.

### Algorithm 1 Abstraction Optimization (Section III-C)

---

**Input:** DNN architecture  $\mathcal{G} = \langle \mathcal{V}, \mathcal{E} \rangle$ , defective node  $n_0 \in \mathcal{V}$ , nodes to impose fix  $\mathcal{V}_{\text{fix}} \subseteq \mathcal{V}$

- 1:  $s \leftarrow 1, \gamma_s \leftarrow 0.9, \gamma_c \leftarrow 0.1, \text{minstep} \leftarrow 0.1, \text{maxiter} \leftarrow 1000$
- 2:  $c_n \leftarrow (l_n^{\text{valid}} + u_n^{\text{valid}})/2, l_n \leftarrow l_n^{\text{valid}}, u_n \leftarrow u_n^{\text{valid}}, \forall n \in \mathcal{V}_{\text{fix}}$
- 3: **for**  $i = 1$  to **maxiter** **do**
- 4:   **for**  $n \in \mathcal{V}_{\text{fix}}$  **do**
- 5:      $\text{loss} \leftarrow \mathcal{L}_{n_0}^{\text{precond}}(\{l_{n'}, u_{n'}\}_{n' \in \mathcal{V}_{\text{fix}}})$
- 6:      $c_n \leftarrow c_n - \gamma_c \max\{|c_n|, \text{minstep}\} \text{sgn}(\nabla_{c_n} \text{loss})$
- 7:      $(l_n, u_n) \leftarrow (c_n - \frac{s(u_n^{\text{valid}} - l_n^{\text{valid}})}{2}, c_n + \frac{s(u_n^{\text{valid}} - l_n^{\text{valid}})}{2})$
- 8:      $(l_n, u_n) \leftarrow (\max\{l_n, l_n^{\text{valid}}\}, \min\{u_n, u_n^{\text{valid}}\})$
- 9:   **end for**
- 10:   **if**  $\mathcal{L}_{n_0}^{\text{precond}}(\{l_n, u_n\}_{n \in \mathcal{V}_{\text{fix}}}) < 0$  **then**
- 11:     **return**  $\{l_n, u_n\}_{n \in \mathcal{V}_{\text{fix}}}$      // Find precondition fix
- 12:   **end if**
- 13:    $s \leftarrow \gamma_s \cdot s$
- 14: **end for**
- 15: **return** “failed”     // Failed to find precondition fix

---

Equation (5) guarantees the validity of the precondition as a fix.

For any  $\{l_n, u_n\}_{n \in \mathcal{V}_{\text{fix}}}$ , thanks to RANUM's static analysis framework, we can compute induced intervals of defective node  $n_0$ , and thus compute the loss value  $\mathcal{L}_{n_0}^{\text{precond}}$ .

As shown in Algorithm 1, we propose the technique of **abstraction optimization** to effectively and approximately solve this optimization. Our technique works iteratively. In the first iteration, we set span  $s = 1$ , and in the subsequent iterations, we reduce the span  $s$  exponentially as shown in Line 13 where hyperparameter  $\gamma_s = 0.9$ . Inside each iteration, for each node to impose precondition  $n \in \mathcal{V}_{\text{fix}}$ , we use the interval center  $c_n = (l_n + u_n)/2$  as the optimizable variable and compute the *sign* of its gradient:  $\text{sgn}(\nabla_{c_n} \text{loss})$ . We use this gradient sign to update each  $c_n$  toward reducing the loss value in Line 6. Then, we use  $c_n$  and the span  $s$  to recover the actual interval in Line 7 and clip  $l_n$  and  $u_n$  by the valid range  $[l_n^{\text{valid}}, u_n^{\text{valid}}]$  in Line 8. At the end of this iteration, for updated  $l_n$  and  $u_n$ , we compute  $\mathcal{L}_{n_0}^{\text{precond}}(\{l_n, u_n\}_{n \in \mathcal{V}_{\text{fix}}})$  to check whether the precondition is a fix. If so, we terminate; otherwise, we proceed to the next iteration. We note that *if the algorithm finds a precondition, the precondition is guaranteed to be a valid fix* by the soundness nature of our static analysis framework and the definition of  $\mathcal{L}_{n_0}^{\text{precond}}$ . When no feasible precondition is found within  $\text{maxiter} = 1000$  iterations, we terminate the algorithm and report “failed to find the fix”.

*Remark.* The key ingredient in the technique is the gradient-sign-based update rule (shown in Line 6), which is much more effective than normal gradient descent for two reasons. (1) Our update rule can get rid of gradient explosion and vanishing problems. For early optimization iterations, the span  $s$  is large and interval bounds are generally coarse, resulting in too large or too small gradient magnitude. For example, the input range for Log could be  $[1, 10^{10}]$  where gradient can be  $10^{-10}$ , resulting in almost negligible gradient updates. In contrast, our update rule leverages the gradient sign, which always points to the correct gradient direction. The update step size in our rule is the maximum of current magnitude  $|c_n|$  and  $\text{minstep}$  to avoid stagnation. (2) Our update rule mitigates the gradient

magnitude discrepancy of different  $c_n$ . At different locations, the nodes in DNNs can have diverse value magnitudes that are not aligned with their gradient magnitudes, making gradient optimization challenging. Therefore, we use this update rule to solve the challenge, where the update magnitude depends on the value magnitude ( $|c_n|$ ) instead of gradient magnitude ( $\nabla_{c_n} \text{loss}$ ). We empirically compare our technique with standard gradient descent in Section IV-C.

#### IV. EXPERIMENTAL EVALUATION

We conduct a systematic experimental evaluation to answer the following research questions.

- RQ1** For tasks already supported by existing state-of-the-art (SOTA) tools (tasks ① and ②a), how much more effective and efficient is RANUM compared to these SOTA tools?
- RQ2** For feasibility confirmation via *generating failure-exhibiting system tests* (task ②), how much more effectively and efficiently can RANUM confirm potential numerical defects compared to baseline approaches?
- RQ3** For *suggesting fixes* (task ③), how much more efficient and effective is RANUM in terms of guarding against numerical failures compared to baseline approaches and developers’ fixes, respectively?

For RQ1, we compare RANUM with all SOTA tools. For RQ2 and RQ3, RANUM is the first approach to the best of our knowledge, so we compare RANUM with baseline approaches (constructed by leaving our novel techniques out of RANUM) and developers’ fixes. We conduct the evaluation on the GRIST benchmarks [55], being the largest dataset of real-world DNN numerical defects to our knowledge. The benchmarks contain 63 real-world DL programs with numerical defects collected from previous studies and GitHub. Each program contains a DNN architecture, and each architecture has one or more numerical defects. There are 79 real numerical defects in total.

We perform our evaluation on a Linux workstation with a 24-core Xeon E5-2650 CPU running at 2.20 GHz. Throughout the evaluation, we stop the execution after reaching 30 min limit by following the evaluation setup by the most recent related work [55].

##### A. RQ1: Comparison with SOTA Tools

For two tasks, existing tools can provide automatic support: potential-defect detection (task ①) where the SOTA tool is DEBAR [62], and failure-exhibiting unit test generation (task ②a) where the SOTA tool is GRIST [55]. We compare RANUM with these tools on their supported tasks, respectively.

**Comparison with DEBAR.** RANUM successfully detects all 79 true defects and DEBAR detects only 48 true defects according to both our evaluation and the literature [55]. Hence, RANUM detects 64.58% more true defects than DEBAR. In terms of efficiency, DEBAR and RANUM have similar running time, and both finish in 3 s per case.

We manually inspect the cases where DEBAR fails but RANUM succeeds. They correspond to DL programs written

with the PyTorch library, which generates dynamic computational graphs that DEBAR cannot handle. In contrast, RANUM provides effective static analysis support for dynamic computational graphs thanks to our backward fine-grained node labeling technique (Section III-A) that is capable of disambiguating the control flow within dynamic graphs.

**Comparison with GRIST.** Results are shown in Table I. Since both RANUM and GRIST have a randomness component where RANUM uses random initialization and GRIST relies on DNN’s randomly initialized weights, we repeat both approaches for 10 runs, record the total number of times where a failure-exhibiting unit test is generated, and the average execution time per run. RANUM succeeds in *all* cases and *all* repeated runs, and GRIST fails to generate such unit test in 24 out of 790 runs (i.e., 96.96% success rate). RANUM has 6.66 s average execution time and is 17.32X faster than GRIST.

The superior effectiveness and efficiency of RANUM are largely due to the existence of random initialization as introduced in Section III-B. We observe that since GRIST always takes initial model weights and inference input as the starting point to update from, the generated unit test is just slightly changed from initial weights and input, being hard to expose the target numerical defect. In contrast, RANUM uses random initialization to explore a much larger space and combines gradient-based optimization to locate the failure-exhibiting instances from the large space. We also evaluate the pure random strategy that uses only random initialization without gradient-based optimization, and such strategy fails in 30 runs, being inferior to both RANUM and GRIST, implying that both random initialization and gradient-based optimization are important. Among all the 79 cases, RANUM is slower than GRIST on only one case (28a). For this case, we find the default inference input loaded by the DNN program (used by GRIST) is not far from a failure-exhibiting one, but a randomly sampled inference input (used by RANUM) is usually far from that. Hence, RANUM takes more iterations to find a failure-exhibiting inference input by the nature of gradient-based optimization.

##### B. RQ2: Feasibility Confirmation via System Test Generation

In task ②, RANUM confirms the feasibility of potential numerical defects by generating failure-exhibiting system tests.

**Baseline.** Since RANUM is the first approach for this task, we do not compare with existing literature and propose one random-based approach (named “Random” hereinafter) as the baseline. In “Random”, we first generate a failure-exhibiting unit test with random sampling. If there is any sample that triggers a failure, we stop and keep the inference example part as the desired  $x_{\text{infer}}$ . Then, we generate  $x_{\text{train}}$  again by random sampling. If any sample, when used for training, could induce model weights  $w$  that cause a numerical failure when using  $x_{\text{infer}}$  as the inference input, we keep that sample as  $x_{\text{train}}$  and terminate. If and only if both  $x_{\text{infer}}$  and  $x_{\text{train}}$  are found, we count this run as a “success” one for “Random”.



TABLE I

(RQ1) RESULTS OF TASK ②A (FAILURE-EXHIBITING UNIT TEST GENERATION) WITH RANUM AND GRIST [55]. C IS THE NUMBER OF RUNS WHERE NUMERICAL FAILURES ARE TRIGGERED IN 10 REPEATED RUNS, T IS THE AVERAGE EXECUTION TIME PER RUN, AND  $\uparrow T$  IS THE AVERAGE TIME IMPROVEMENT ACHIEVED BY RANUM COMPARED TO GRIST.

Case ID	RANUM			GRIST			Case ID	RANUM			GRIST			Case ID	RANUM			GRIST					
	C	T	$\uparrow T$	C	T	$\uparrow T$		C	T	$\uparrow T$	C	T	$\uparrow T$		C	T	$\uparrow T$	C	T	$\uparrow T$			
1	10	9.01	1.20 X	10	10.77	166b	10	0.21	20.85 X	10	4.42	32	10	0.06	27.93 X	10	1.77	47	10	0.06	32.51 X	10	1.87
2a	10	0.02	9.75 X	10	0.24	16c	10	0.25	17.54 X	10	4.43	33	10	0.06	33.63 X	10	1.91	48a	10	0.38	3.06 X	10	1.17
2b	10	0.03	614.68 X	10	16.54	17	10	439.19	+\infty	0	-	34	10	0.06	33.95 X	10	1.90	48b	10	0.15	7.12 X	10	1.10
3	10	0.02	432.11 X	10	8.67	18	10	0.02	1040.46 X	10	22.17	35a	10	0.44	61.76 X	10	27.33	49a	10	0.49	41.09 X	10	20.07
4	10	0.01	1.00 X	10	0.01	19	10	0.16	689.66 X	10	107.78	35b	10	0.45	819.02 X	10	364.86	49b	10	0.50	612.22 X	10	307.24
5	10	0.05	6.48 X	10	0.34	20	10	0.16	3237.27 X	10	511.06	36a	10	0.44	41.80 X	10	18.58	50	10	0.16	781.02 X	10	126.80
6	10	0.84	5.20 X	10	4.38	21	10	0.16	259.73 X	10	42.09	36b	10	0.46	783.16 X	10	362.41	51	10	1.88	671.55 X	3	1263.04
7	10	0.87	4.54 X	10	3.96	22	10	0.94	1518.43 X	10	1433.12	37	10	0.06	38.34 X	10	2.39	52	10	0.15	336.37 X	10	50.59
8	10	0.86	4.63 X	10	3.99	23	10	0.01	157.72 X	10	1.88	38	10	0.06	34.50 X	10	1.94	53	10	0.05	36.64 X	10	1.92
9a	10	0.20	11.03 X	10	2.22	24	10	0.81	40.72 X	10	33.05	39a	10	0.43	42.79 X	10	18.30	54	10	0.05	36.76 X	10	1.83
9b	10	0.14	14.46 X	10	2.09	25	10	0.04	1271.88 X	10	44.70	39b	10	0.43	843.66 X	10	362.22	55	10	0.82	44.63 X	10	36.63
10	10	0.17	228.42 X	10	39.64	26	10	0.05	37.96 X	10	2.00	40	10	0.04	1995.27 X	10	85.97	56	10	0.06	35.04 X	10	1.93
11a	10	0.15	27.58 X	10	4.26	27	10	0.01	185.61 X	10	1.91	41	10	0.04	1967.23 X	10	86.36	57	10	0.01	177.45 X	10	1.88
11b	10	0.13	34.75 X	10	4.38	28a	10	24.37	-13.30 X	10	1.83	42	10	0.05	1934.84 X	10	87.89	58	10	0.83	12.01 X	10	9.95
11c	10	0.11	4499.86 X	10	516.13	28b	10	24.17	7.28 X	10	176.02	43a	10	0.48	35.63 X	10	16.96	59	10	0.02	105.40 X	10	1.94
12	10	0.26	135.94 X	10	34.69	28c	10	0.12	8.69 X	10	1.02	43b	10	0.45	4008.93 X	10	1800.00	60	10	0.15	221.97 X	10	34.19
13	10	0.01	1.10 X	10	0.01	28d	10	0.12	1518.28 X	10	176.02	44	10	0.27	579.29 X	10	155.38	61	10	0.35	53.29 X	10	18.78
14	10	0.80	107.96 X	10	86.23	29	10	0.89	16.83 X	10	14.98	45a	10	0.16	417.25 X	10	68.08	62	10	1.85	72.19 X	10	133.62
15	10	1.71	5.95 X	10	10.18	30	10	0.16	222.12 X	10	35.61	45b	10	0.88	14.69 X	10	12.98	63	10	2.06	117.12 X	10	240.68
16a	10	0.12	34.24 X	10	4.02	31	10	3.13	2.41 X	3	7.54	46	10	0.01	168.39 X	10	1.88	Tot: 79	790	6.66	17.32 X	766	115.30

TABLE II

(RQ2) RESULTS OF TASK ② (FAILURE-EXHIBITING SYSTEM TEST GENERATION) WITH RANUM AND RANDOM (BASELINE). C IS THE TOTAL NUMBER OF RUNS WHERE NUMERICAL FAILURES ARE TRIGGERED IN 10 REPEATED RUNS. T IS THE AVERAGE EXECUTION TIME PER RUN.

Case ID	RANUM		Random		Case ID	RANUM		Random		Case ID	RANUM		Random		Case ID	RANUM		Random						
	C	T	C	T		C	T	C	T		C	T	C	T		C	T	C	T	C	T			
1	0	9.01	0	1806.13	13	10	0.01	10	0.01	27	10	0.01	10	0.01	38	1	0.13	0	1800.65	49b	10	0.50	8	364.09
2a	10	0.03	10	0.06	14	10	12.60	10	0.50	28a	0	24.37	0	1920.29	39a	10	0.43	1	1623.72	50	10	4.89	10	0.16
2b	10	0.03	10	0.06	15	0	1.71	0	2107.24	28b	0	24.17	0	1911.26	39b	10	0.43	8	364.10	51	10	49.12	10	2.10
3	10	0.02	10	0.05	16a	10	0.12	10	0.75	28c	10	0.12	10	0.53	40	10	0.06	10	0.02	52	10	4.87	10	0.15
4	10	0.01	10	0.01	16b	10	0.21	0	1834.44	28d	10	0.12	10	0.48	41	10	0.06	10	0.02	53	10	0.07	10	0.03
5	10	0.05	10	0.06	16c	10	0.25	0	1831.67	29	10	0.89	10	11.96	42	10	0.06	10	0.02	54	10	0.07	10	0.02
6	10	0.84	10	12.42	17	10	549.98	10	235.11	30	10	4.88	10	0.14	43a	10	0.48	1	1623.71	55	10	0.82	10	12.79
7	10	0.87	10	12.51	18	10	0.02	10	0.05	31	10	14.62	10	9.31	43b	10	0.45	9	184.14	56	10	0.07	10	0.03
8	10	0.86	10	12.37	19	10	4.88	10	0.16	32	10	0.08	10	0.03	44	10	0.27	10	1.36	57	10	0.01	8	360.01
9a	10	0.20	7	541.25	20	10	4.88	10	0.14	33	10	0.07	10	0.02	45a	10	4.89	10	0.15	58	10	0.83	10	12.28
9b	10	0.14	10	1.39	21	10	4.89	10	0.14	34	10	0.42	10	0.20	45b	10	0.88	10	12.27	59	10	0.02	10	0.05
10	10	4.90	10	0.16	22	10	500.10	0	1801.60	35a	10	0.44	10	4.01	46	10	0.01	10	0.01	60	10	4.88	10	0.16
11a	10	0.15	10	0.72	23	10	0.01	10	0.01	35b	10	0.45	10	4.22	47	10	0.08	10	0.03	61	10	9.84	10	0.93
11b	10	0.13	10	0.76	24	10	0.81	10	12.52	36a	10	0.44	1	1623.76	48a	10	9.89	10	0.90	62	10	48.86	10	2.73
11c	10	0.11	10	0.74	25	10	0.04	10	0.15	36b	10	0.46	3	1263.79	48b	10	4.88	10	0.15	63	10	49.06	10	2.15
12	10	0.26	10	0.72	26	10	0.07	10	0.03	37	2	0.07	2	1440.50	49a	10	0.49	1	1623.88	Tot: 79	733	17.31 (19.30X)	649	334.14

For each defect, due to the randomness of the model’s initial weights, we repeat both RANUM and “Random” for 10 runs. Both approaches use the same set of random seeds.

**Evaluation Result.** Results are in Table II. We observe that RANUM succeeds in  $733/(79 \times 10) = 92.78\%$  runs and the baseline “Random” succeeds in  $649/(79 \times 10) = 82.15\%$  runs. Moreover, RANUM spends only 17.31s time on average per run, which is a 19.30X speedup compared to “Random”. We also observe that RANUM is more reliable across repeated runs. There are only 6 cases with unsuccessful repeated runs in RANUM, but there are 19 such cases in “Random”. Hence, RANUM is substantially more effective, efficient, and reliable for generating system tests than the baseline.

**Discussion.** The high effectiveness of RANUM mainly comes from the advantage of gradient-guided search compared with random search. As described in Section III-B, RANUM leverages both first-order gradients (in step a) and second-order derivatives (in step b) to guide the search of system tests. In contrast, “Random” uses random sampling hoping that failure-exhibiting training examples can emerge after sufficient sampling. Hence, when such training examples are relatively rare in the whole valid input space, “Random” is less effective. We conduct an ablation study (in Suppl. F) for showing that RANUM improves over “Random” in both steps inside RANUM.

**Failing-Case Analysis.** We study all six defects where RANUM may fail and have the following findings. (1) For four defects (Case IDs 1, 15, 37, and 38), the architecture is

TABLE III

(RQ3) RESULTS OF TASK ③ (FIX SUGGESTION) UNDER THREE IMPOSING LOCATION SPECIFICATIONS WITH RANUM AND TWO BASELINES (RANUM-E AND GD). # IS THE NUMBER OF FIXES FOUND. “Time (s)” IS THE TOTAL RUNNING TIME FOR ALL 79 CASES.

Imposing Locations	RANUM		RANUM-E		GD	
	#	Time (s)	#	Time (s)	#	Time (s)
Weight + Input	79	54.23	78	540.13	57	188.63
Weight	72	58.47	71	581.86	43	219.28
Input	37	924.74	37	3977.30	29	952.19

challenging for gradient-based optimization, e.g., due to the `Min/Max/Softmax` operators that provide little or no gradient information. We leave it as future work to solve these cases, likely in need of dynamically detecting operators with vanishing gradients and reconstructing the gradient flow. (2) Two defects (Case IDs 28a and 28b) correspond to those caused by `Div` operators where only a close-to-zero divisor can trigger a numerical failure. Hence, for operators with narrow invalid ranges, RANUM may fail to generate failure-exhibiting system tests.

### C. RQ3: Fix Suggestion

In task ③, RANUM suggests fixes for numerical defects. We compare RANUM with fixes generated by baseline approaches and developers’ fixes.

#### 1) Comparison between RANUM and Baselines:

RANUM is the first approach for this task, and we propose two baseline approaches to compare with. (1) RANUM-E: this approach changes the abstraction domain of RANUM from interval with tensor partitioning to standard interval.

To some degree, RANUM-E represents the methodology of conventional static analysis tools that use standard interval domain for abstraction and search of effective fixes. (2) GD: this approach uses standard gradient descent for optimization instead of the abstraction optimization technique in RANUM.

**Evaluation Protocol.** We evaluate whether each approach can generate fixes that eliminate *all* numerical defects for the DNN architecture under analysis given imposing locations. We consider three types of locations: on both weight and input nodes, on only weight nodes, and on only input nodes. In practice, model providers can impose fixes on weight nodes by clipping weights after a model is trained; and users can impose fixes on input nodes by clipping their inputs before loading them into the model. Since all approaches are deterministic, for each case we run only once. We say that the fix eliminates all numerical defects if and only if (1) the RANUM static analysis framework cannot detect any defects from the fixed architecture; and (2) 1,000 random samples cannot trigger any numerical failures after imposing the fix.

**Evaluation Result.** We report the statistics, including the number of successful cases among all the 79 cases and the total running time, in Table III. From the table, we observe that on all the three imposing location settings, RANUM always succeeds in most cases and spends much less time. For example, when fixes can be imposed on both weights and input nodes, RANUM succeeds on *all* cases with a total running time 54.23 s. In contrast, RANUM-E requires  $> 10\times$  time, and GD succeeds in only 72.15% cases. Hence, RANUM is substantially more effective and efficient for suggesting fixes compared to baseline approaches.

Since RANUM is based on iterative refinement (see Algorithm 1), we study the number of iterations needed for finding the fix. When fixes can be imposed on both weight and input nodes, where RANUM succeeds on all the 79 cases, the average number of iterations is 29.80, the standard deviation is 14.33, the maximum is 53, and the minimum is 2. Hence, when RANUM can find the fix, the number of iterations is small, coinciding with the small total running time 54.23 s.

**Discussion.** The two baseline approaches can be viewed as ablated versions of RANUM. Comparing RANUM and GD, we conclude that the technique of abstraction optimization substantially improves the effectiveness and also improves the efficiency. Comparing RANUM and RANUM-E, we conclude that the interval abstraction with tensor partitioning as the abstraction domain substantially improves the efficiency and also improves the effectiveness.

From Table III, it is much easier to find the fix when imposing locations are weight nodes compared to input nodes. Since model providers can impose fixes on weights and users impose on inputs, this finding implies that fixing numerical defects on the providers' side may be more effective than on the users' side.

2) *Comparison between RANUM and Developers' Fixes:* We conduct an empirical study to compare the fixes generated by RANUM and by the developers.

**Evaluation Protocol.** We manually locate GitHub repositories from which the GRIST benchmarks are constructed. Among the 79 cases, we find the repositories for 53 cases on GitHub and we study these cases. We locate the developers' fixes of the numerical defects by looking at issues and follow-up pull requests. Since RANUM suggests different fixes for different imposing locations, for each case we first determine the imposing locations from the developer's fix, and then compare with RANUM's fix for these locations.

RANUM fixes are on the computational graph and developers' fixes are in the source code, so we determine to conduct code-centered comparison: RANUM fixes are considered feasible only when the fixes can be easily implemented by code (within 10 lines of code) given that developers' fixes are typically small, usually in 3-5 lines of code. In particular, our comparison is based on two criteria: (1) which fix is sound on any valid input; (2) if both are sound, which fix hurts less to model performance and utility (based on the span of imposed precondition, the larger span the less hurt). Two authors independently classify the comparison results for each case and discuss the results to reach a consensus.

**Results.** We categorize the comparison results as below.

- A (30 cases) *Better than developers' fixes or no available developer's fix.* Developers either propose no fixes or use heuristic fixes, such as reducing the learning rate or using the mean value to reduce the variance. These fixes may work in practice but are unsound, i.e., cannot rigorously guarantee the elimination of the numerical defect for any training or inference data. In contrast, RANUM generates better fixes since these fixes rigorously eliminate the defect.
- B (7 cases) *Equivalent to developers' fixes.* Developers and RANUM suggest equivalent or highly similar fixes.
- C (13 cases) *No need to fix.* For these cases, there is no need to fix the numerical defect in the given architecture. There are mainly three reasons. (1) The DNN is used in the whole project with fixed weights or test inputs. As a result, although the architecture contains defects, no system failure can be caused. (2) The architecture is injected a defect as a test case for automatic tools, such as a test architecture in the `TensorFuzz` [28] repository. (3) The defect can be hardly exposed in practice. For example, the defect is in a `Div` operator where the divisor needs to be very close to zero to trigger a divide-by-zero failure, but such situation hardly happens in practice since the divisor is randomly initialized.
- D (3 cases) *Inferior than developers' fixes or RANUM-generated fixes are impractical.* In two cases, RANUM-generated fixes are inferior to developers' fixes. Human developers observe that the defective operator is `Log`, and its input is non-negative. So they propose to add  $10^{-6}$  to the input of `Log` as the fix. In contrast, RANUM can generate only a clipping-based fix, e.g., clipping the input if it is less than  $10^{-6}$ . When the input is small, RANUM's fix interrupts the gradient flow from output to input while the human's fix maintains it. As a result, the human's fix does less hurt to the model's trainability and is better than

RANUM’s fix. In another case, the RANUM-generated fix imposes a small span for some model weights (less than 0.1 for each component of that weight node). Such a small weight span strongly limits the model’s expressiveness and utility. We leave it as the future work to solve these limitations.

From the comparison results, we can conclude that for the 40 cases where numerical defects are needed to be fixed (excluding case C), RANUM suggests equivalent or better fixes than human developers in 37 cases. Therefore, RANUM is comparably effective as human developers in terms of suggesting numerical-defect fixes, and is much more efficient since RANUM is an automatic approach.

**Guidelines for Users.** We discuss two practical questions for RANUM users. (1) *Does RANUM hurt model utility, e.g., inference accuracy?* If no training or test data ever exposes a numerical defect, RANUM does not confirm a defect and hence no fix is generated and there is no hurt to the utility. If RANUM confirms numerical defects, whether the fix affects the utility depends on the precondition-imposing locations. If imposing locations can be freely selected, RANUM tends to impose the fix right before the vulnerable operator, and hence the fix does not reduce inference performance. The reason is that the fix changes (by clipping) the input only when the input falls in the invalid range of the vulnerable operator. In practice, if the imposing locations cannot be freely selected and need to follow developers’ requirements, our preceding empirical study shows that, in only 3 out of 40 cases, compared with developers’ fixes, our fixes incur larger hurt to the inference or training performance of the architecture. (2) *Should we always apply RANUM to fix any architecture?* We can always apply RANUM to fix any architecture since RANUM fixes do not visibly alter the utility in most cases. Nonetheless, in deployment, we recommend first using RANUM to confirm defect feasibility. If there is no such failure-exhibiting system test, we may not need to fix the architecture; otherwise, we use RANUM to generate fixes.

## V. RELATED WORK

**Understanding and Detecting Defects in DNNs.** Discovering and mitigating defects and failures in DNN based systems is an important research topic [61, 33, 14]. Following the taxonomy in previous work [12], DNN defects are at four levels from bottom to top. (1) Platform-level defects. Defects can exist in real-world DL compilers and libraries. Approaches exist for understanding, detecting, and testing against these defects [50, 38, 43, 53]. (2) Architecture-level defects. *Our work focuses on numerical defects, being one type of architecture-level defects.* Automatic detection and localization approaches [51, 20] exist for other architecture-level defects such as suboptimal structure, activation function, and initialization and shape mismatch [11]. (3) Model-level defects. Once a model is trained, its defects can be viewed as violations of desired properties as discussed by Zhang et al. [56]. Some example defects are correctness [45, 9], robustness [49], and fairness [58] defects. (4) Interface-level

defects. DNN-based systems, when deployed as services, expose interaction interfaces to users where defects may exist, as shown by empirical studies on real-world systems [12, 47, 48].

**Testing and Debugging for DNNs.** A rich body of work exists for testing and debugging DNN defects [56]. Some representatives are DeepXplore [32] and DeepGauge [23]. Recent work enables automatic model debugging and repair via consistency checking [52], log checking [60], spectrum analysis [35], or analyzer-guided synthesis [42].

**DNN Static Analysis.** Another solution for eliminating DNN defects is conducting static analysis to rigorously guarantee the non-existence of defects [17, 2]. Although DNNs essentially conduct numerical computations, traditional tools of numerical analysis [10, 40] are inefficient for DNN analysis due to lack of support for multi-dimensional tensor computations. Recently, static analysis tools customized for DNNs are emerging, mainly focusing on proposing tighter abstractions [6, 27, 30] or incorporating abstractions into training to improve robustness [16, 26, 63]. Besides robustness, static analysis has also been applied to rigorously bound model difference [31]. Our approach includes a static analysis framework customized for numerical-defect detection and fixing.

**Detecting and Exposing Numerical Defects in DNNs.** Despite the widespread existence of numerical defects in real-world DNN-based systems [61, 12, 14], only a few automatic approaches exist for detecting and exposing these defects. To the best of our knowledge, DEBAR [62] and GRIST [55] are the only two approaches. We discuss and compare RANUM with both approaches extensively in Sections III and IV.

## VI. CONCLUSION

In this paper, we have presented a novel automatic approach named RANUM for reliability assurance of DNNs against numerical defects. RANUM supports detection of potential numerical defects, confirmation of potential-defect feasibility, and suggestion of defect fixes. RANUM includes multiple novel extensions and optimizations upon existing tools, and includes three novel techniques. Our extensive evaluation on real-world DNN architectures has demonstrated high effectiveness and efficiency of RANUM compared to both the state-of-the-art approaches and developers’ fixes.

**Data Availability.** All artifacts including the tool source code and experiment logs are available and actively maintained at <https://github.com/1lylly/RANUM>. The supplemental materials containing proofs, hyperparameters, and experiments are available in the arXiv version [18].

## ACKNOWLEDGEMENTS

This work is sponsored by the National Natural Science Foundation of China under Grant No. 62161146003, the National Key Research and Development Program of China under Grant No. 2019YFE0198100, the Innovation and Technology Commission of HKSAR under Grant No. MHP/055/19, and the Tencent Foundation/XPLORER PRIZE.

## REFERENCES

- [1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. J. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Józefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. G. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. A. Tucker, V. Vanhoucke, V. Vasudevan, F. B. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, “TensorFlow: Large-scale machine learning on heterogeneous distributed systems,” *CoRR*, vol. abs/1603.04467, 2016.
- [2] A. Albarghouthi, “Introduction to neural network verification,” *Foundations and Trends in Programming Languages*, vol. 7, no. 1-2, pp. 1–157, 2021.
- [3] Y. Bengio, N. Léonard, and A. C. Courville, “Estimating or propagating gradients through stochastic neurons for conditional computation,” *CoRR*, vol. abs/1308.3432, 2013.
- [4] N. D. Q. Bui, Y. Yu, and L. Jiang, “InferCode: Self-supervised learning of code representations by predicting subtrees,” in *43rd IEEE/ACM International Conference on Software Engineering, ICSE*. IEEE, 2021, pp. 1186–1197.
- [5] P. Cousot and R. Cousot, “Static determination of dynamic properties of generalized type unions,” in *ACM Conference on Language Design for Reliable Software, LDRS*. ACM, 1977, pp. 77–94.
- [6] T. Gehr, M. Mirman, D. Drachler-Cohen, P. Tsankov, S. Chaudhuri, and M. Vechev, “AI2: safety and robustness certification of neural networks with abstract interpretation,” in *39th IEEE Symposium on Security and Privacy, SP*. IEEE, 2018, pp. 3–18.
- [7] X. Glorot, A. Bordes, and Y. Bengio, “Deep sparse rectifier neural networks,” in *14th International Conference on Artificial Intelligence and Statistics, AISTATS*. JMLR.org, 2011, pp. 315–323.
- [8] I. J. Goodfellow, J. Shlens, and C. Szegedy, “Explaining and harnessing adversarial examples,” in *3rd International Conference on Learning Representations, ICLR*. OpenReview.net, 2015.
- [9] A. Guerriero, R. Pietrantuono, and S. Russo, “Operation is the hardest teacher: estimating DNN accuracy looking for mispredictions,” in *43rd IEEE/ACM International Conference on Software Engineering, ICSE*. IEEE, 2021, pp. 348–358.
- [10] A. Gurfinkel, T. Kahsai, A. Komuravelli, and J. A. Navas, “The SeaHorn verification framework,” in *27th International Conference on Computer Aided Verification, CAV*. Springer, 2015, pp. 343–361.
- [11] M. Hattori, S. Sawada, S. Hamaji, M. Sakai, and S. Shimizu, “Semi-static type, shape, and symbolic shape inference for dynamic computation graphs,” in *4th ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, 2020, pp. 11–19.
- [12] N. Humbatova, G. Jahangirova, G. Bavota, V. Riccio, A. Stocco, and P. Tonella, “Taxonomy of real faults in deep learning systems,” in *42nd ACM/IEEE International Conference on Software Engineering, ICSE*. IEEE, 2020, pp. 1110–1121.
- [13] N. Jay, N. Rotman, B. Godfrey, M. Schapira, and A. Tamar, “A deep reinforcement learning perspective on Internet congestion control,” in *36th International Conference on Machine Learning, ICML*. PMLR, 2019, pp. 3050–3059.
- [14] E. Kloberdanz, K. G. Kloberdanz, and W. Le, “DeepStability: A study of unstable numerical methods and their solutions in deep learning,” in *44th International Conference on Software Engineering, ICSE*. ACM, 2022, pp. 586–597.
- [15] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “ImageNet classification with deep convolutional neural networks,” in *Advances in Neural Information Processing Systems 25, NIPS*, 2012, pp. 1106–1114.
- [16] L. Li, Z. Zhong, B. Li, and T. Xie, “Robustra: Training provable robust neural networks over reference adversarial space,” in *28th International Joint Conference on Artificial Intelligence, IJCAI*. ijcai.org, 2019, pp. 4711–4717.
- [17] L. Li, T. Xie, and B. Li, “SoK: Certified robustness for deep neural networks,” in *44th IEEE Symposium on Security and Privacy, SP*. IEEE, 2023, pp. 94–115.
- [18] L. Li, Y. Zhang, L. Ren, Y. Xiong, and T. Xie, “Reliability assurance for deep neural network architectures against numerical defects,” *CoRR*, vol. abs/2302.06086, 2023. [Online]. Available: <https://arxiv.org/pdf/2302.06086.pdf>
- [19] Y. Li and Y. Yuan, “Convergence analysis of two-layer neural networks with ReLU activation,” in *Advances in Neural Information Processing Systems 30, NIPS*, 2017, pp. 597–607.
- [20] C. Liu, J. Lu, G. Li, T. Yuan, L. Li, F. Tan, J. Yang, L. You, and J. Xue, “Detecting TensorFlow program bugs in real-world industrial environment,” in *36th IEEE/ACM International Conference on Automated Software Engineering, ASE*. IEEE, 2021, pp. 55–66.
- [21] F. Liu, G. Li, B. Wei, X. Xia, Z. Fu, and Z. Jin, “A unified multi-task learning model for AST-level and token-level code completion,” *Empirical Software Engineering*, vol. 27, no. 4:91, 2022.
- [22] Z. Liu, P. Luo, X. Wang, and X. Tang, “Deep learning face attributes in the wild,” in *2015 IEEE International Conference on Computer Vision, ICCV*. IEEE, 2015, pp. 3730–3738.
- [23] L. Ma, F. Juefei-Xu, F. Zhang, J. Sun, M. Xue, B. Li, C. Chen, T. Su, L. Li, Y. Liu, J. Zhao, and Y. Wang, “DeepGauge: Multi-granularity testing criteria for deep learning systems,” in *33rd ACM/IEEE International Conference on Automated Software Engineering, ASE*. ACM, 2018, pp. 120–131.
- [24] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu, “Towards deep learning models resistant to adversarial attacks,” in *6th International Conference on Learning Representations, ICLR*. OpenReview.net, 2018.
- [25] P. McCausland, “Self-driving uber car that hit and killed woman did not recognize that pedestrians jaywalk,” 2022. [Online]. Available: <https://www.nbcnews.com/tech-news/n1079281>
- [26] M. Mirman, T. Gehr, and M. T. Vechev, “Differentiable abstract interpretation for provably robust neural networks,” in *35th International Conference on Machine Learning, ICML*. PMLR, 2018, pp. 3575–3583.
- [27] M. N. Müller, G. Makarchuk, G. Singh, M. Püschel, and M. Vechev, “PRIMA: general and precise neural network certification via scalable convex hull approximations,” *the ACM on Programming Languages*, vol. 6, no. POPL, pp. 1–33, 2022.
- [28] A. Odena, C. Olsson, D. Andersen, and I. Goodfellow, “TensorFuzz: Debugging neural networks with coverage-guided fuzzing,” in *36th International Conference on Machine Learning, ICML*. PMLR, 2019, pp. 4901–4911.
- [29] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Z. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, “PyTorch: An imperative style, high-performance deep learning library,” in *Advances in Neural Information Processing Systems 32, NeurIPS*, 2019, pp. 8024–8035.
- [30] B. Paulsen and C. Wang, “LinSyn: Synthesizing tight linear bounds for arbitrary neural network activation functions,” in *28th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS*. Springer, 2022, pp. 357–376.
- [31] B. Paulsen, J. Wang, J. Wang, and C. Wang, “NeuroDiff: scalable differential verification of neural networks using fine-grained approximation,” in *35th IEEE/ACM International Conference on Automated Software Engineering, ASE*. IEEE, 2020, pp. 784–796.

- [32] K. Pei, Y. Cao, J. Yang, and S. Jana, “DeepXplore: automated whitebox testing of deep learning systems,” *Communications of the ACM*, vol. 62, no. 11, pp. 137–145, 2019.
- [33] H. V. Pham, S. Qian, J. Wang, T. Lutellier, J. Rosenthal, L. Tan, Y. Yu, and N. Nagappan, “Problems and opportunities in training deep learning software systems: An analysis of variance,” in *35th IEEE/ACM International Conference on Automated Software Engineering, ASE*. IEEE, 2020, pp. 771–783.
- [34] L. Powell, “The problem with artificial intelligence in security,” 2022. [Online]. Available: <https://darkreading.com/threat-intelligence/the-problem-with-artificial-intelligence-in-security>
- [35] H. Qi, Z. Wang, Q. Guo, J. Chen, F. Juefei-Xu, L. Ma, and J. Zhao, “ArchRepair: Block-level architecture-oriented repairing for deep neural networks,” *CoRR*, vol. abs/2111.13330, 2021.
- [36] T. Serra, C. Tjandraatmadja, and S. Ramalingam, “Bounding and counting linear regions of deep neural networks,” in *35th International Conference on Machine Learning, ICML*. PMLR, 2018, pp. 4565–4573.
- [37] D. K. Sharma, S. K. Dhurandher, I. Woungang, R. K. Srivastava, A. Mohanane, and J. J. Rodrigues, “A machine learning-based protocol for efficient routing in opportunistic networks,” *IEEE Systems Journal*, vol. 12, no. 3, pp. 2207–2213, 2016.
- [38] Q. Shen, H. Ma, J. Chen, Y. Tian, S. Cheung, and X. Chen, “A comprehensive study of deep learning compiler bugs,” in *29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE*. ACM, 2021, pp. 968–980.
- [39] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. P. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis, “Mastering the game of Go without human knowledge,” *Nature*, vol. 550, no. 7676, pp. 354–359, 2017.
- [40] G. Singh, M. Püschel, and M. T. Vechev, “Fast polyhedra abstract domain,” in *44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL*. ACM, 2017, pp. 46–59.
- [41] G. Singh, T. Gehr, M. Püschel, and M. Vechev, “An abstract domain for certifying neural networks,” *The ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–30, 2019.
- [42] M. Sotoudeh and A. V. Thakur, “Provable repair of deep neural networks,” in *42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI*. ACM, 2021, pp. 588–603.
- [43] F. Tambon, A. Nikanjam, L. An, F. Khomh, and G. Antoniol, “Silent bugs in deep learning frameworks: An empirical study of Keras and TensorFlow,” *CoRR*, vol. abs/2112.13314, 2021.
- [44] The Linux Foundation, “ONNX home,” <https://onnx.ai/>, 2022, accessed: 2023-02-01.
- [45] S. Tizpaz-Niari, P. Cerný, and A. Trivedi, “Detecting and understanding real-world differential performance bugs in machine learning libraries,” in *29th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA*. ACM, 2020, pp. 189–199.
- [46] F. Tramèr, N. Carlini, W. Brendel, and A. Madry, “On adaptive attacks to adversarial example defenses,” in *Advances in Neural Information Processing Systems 33, NeurIPS*, 2020, pp. 1633–1645.
- [47] C. Wan, S. Liu, H. Hoffmann, M. Maire, and S. Lu, “Are machine learning cloud APIs used correctly?” in *43rd IEEE/ACM International Conference on Software Engineering, ICSE*. IEEE, 2021, pp. 125–137.
- [48] C. Wan, S. Liu, S. Xie, Y. Liu, H. Hoffmann, M. Maire, and S. Lu, “Automated testing of software that uses machine learning APIs,” in *44th IEEE/ACM International Conference on Software Engineering, ICSE*. ACM, 2022, pp. 212–224.
- [49] J. Wang, J. Chen, Y. Sun, X. Ma, D. Wang, J. Sun, and P. Cheng, “RobOT: Robustness-oriented testing for deep learning systems,” in *43rd IEEE/ACM International Conference on Software Engineering, ICSE*. IEEE, 2021, pp. 300–311.
- [50] Z. Wang, M. Yan, J. Chen, S. Liu, and D. Zhang, “Deep learning library testing via effective model generation,” in *28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE*. ACM, 2020, pp. 788–799.
- [51] M. Wardat, W. Le, and H. Rajan, “DeepLocalize: Fault localization for deep neural networks,” in *43rd IEEE/ACM International Conference on Software Engineering, ICSE*. IEEE, 2021, pp. 251–262.
- [52] Y. Xiao, I. Beschastnikh, D. S. Rosenblum, C. Sun, S. Elbaum, Y. Lin, and J. S. Dong, “Self-checking deep neural networks in deployment,” in *43rd IEEE/ACM International Conference on Software Engineering, ICSE*. IEEE, 2021, pp. 372–384.
- [53] D. Xie, Y. Li, M. Kim, H. V. Pham, L. Tan, X. Zhang, and M. W. Godfrey, “Leveraging documentation to test deep learning library functions,” *CoRR*, vol. abs/2109.01002, 2021.
- [54] Y. Xiong, Y. Tian, Y. Liu, and S. Cheung, “Toward actionable testing of deep learning models,” *Science China, Information Sciences*, 2022, online first: 2022-09-26.
- [55] M. Yan, J. Chen, X. Zhang, L. Tan, G. Wang, and Z. Wang, “Exposing numerical bugs in deep learning via gradient back-propagation,” in *29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE*. ACM, 2021, pp. 627–638.
- [56] J. M. Zhang, M. Harman, L. Ma, and Y. Liu, “Machine learning testing: Survey, landscapes and horizons,” *IEEE Transactions on Software Engineering*, vol. 48, no. 2, pp. 1–36, 2022.
- [57] K. Zhang, Z. Zhang, Z. Li, and Y. Qiao, “Joint face detection and alignment using multitask cascaded convolutional networks,” *IEEE Signal Processing Letters*, vol. 23, no. 10, pp. 1499–1503, 2016.
- [58] P. Zhang, J. Wang, J. Sun, G. Dong, X. Wang, X. Wang, J. S. Dong, and T. Dai, “White-box fairness testing through adversarial sampling,” in *42nd ACM/IEEE International Conference on Software Engineering, ICSE*. ACM, 2020, pp. 949–960.
- [59] T. Zhang, C. Gao, L. Ma, M. R. Lyu, and M. Kim, “An empirical study of common challenges in developing deep learning applications,” in *30th IEEE International Symposium on Software Reliability Engineering, ISSRE*. IEEE, 2019, pp. 104–115.
- [60] X. Zhang, J. Zhai, S. Ma, and C. Shen, “AutoTrainer: An automatic DNN training problem detection and repair system,” in *43rd IEEE/ACM International Conference on Software Engineering, ICSE*. IEEE, 2021, pp. 359–371.
- [61] Y. Zhang, Y. Chen, S. Cheung, Y. Xiong, and L. Zhang, “An empirical study on TensorFlow program bugs,” in *27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA*. ACM, 2018, pp. 129–140.
- [62] Y. Zhang, L. Ren, L. Chen, Y. Xiong, S. Cheung, and T. Xie, “Detecting numerical bugs in neural network architectures,” in *28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE*. ACM, 2020, pp. 826–837.
- [63] Y. Zhang, A. Albarghouthi, and L. D’Antoni, “Certified robustness to programmable transformations in LSTMs,” in *2021 Conference on Empirical Methods in Natural Language Processing, EMNLP*. Association for Computational Linguistics, 2021, pp. 1068–1083.
- [64] L. Zhu, Z. Liu, and S. Han, “Deep leakage from gradients,” in *Advances in Neural Information Processing Systems 32, NeurIPS*, 2019, pp. 14747–14756.