Detecting Numerical Bugs in Neural Network Architectures

Yuhao Zhang
Key Laboratory of High Confidence Software Technologies, MoE
Department of Computer Science and Technology, Peking University
Beijing, PR China
yuhaoz@cs.wisc.edu

Luyao Ren
Key Laboratory of High Confidence Software Technologies, MoE
Department of Computer Science and Technology, Peking University
Beijing, PR China
rly@pku.edu.cn

Liqian Chen
Key Laboratory of Software Engineering for Complex Systems, College of Computer, National University of Defense Technology
Changsha, PR China
lqchen@nudt.edu.cn

Yingfei Xiong*
Key Laboratory of High Confidence Software Technologies, MoE
Department of Computer Science and Technology, Peking University
Beijing, PR China
xiongyf@pku.edu.cn

Shing-Chi Cheung
Department of Computer Science and Engineering, The Hong Kong University of Science and Technology
Hong Kong, PR China
scc@cse.ust.hk

Tao Xie
Key Laboratory of High Confidence Software Technologies, MoE
Department of Computer Science and Technology, Peking University
Beijing, PR China
taoxie@pku.edu.cn

ABSTRACT

Detecting bugs in deep learning software at the architecture level provides additional benefits that detecting bugs at the model level does not provide. This paper makes the first attempt to conduct static analysis for detecting numerical bugs at the architecture level. We propose a static analysis approach for detecting numerical bugs in neural architectures based on abstract interpretation. Our approach mainly comprises two kinds of abstraction techniques, i.e., one for tensors and one for numerical values. Moreover, to scale up while maintaining adequate detection precision, we propose two abstraction techniques: tensor partitioning and (elementwise) affine relation analysis to abstract tensors and numerical values, respectively. We realize the combination scheme of tensor partitioning and affine relation analysis (together with interval analysis) as DEBAR, and evaluate it on two datasets: neural architectures with known bugs (collected from existing studies) and real-world neural architectures. The evaluation results show that DEBAR outperforms other tensor and numerical abstraction techniques on accuracy without losing scalability. DEBAR successfully detects all known numerical bugs with no false positives within 1.7–2.3 seconds per architecture. On the real-world architectures, DEBAR reports 529 warnings within 2.6–135.4 seconds per architecture, where 299 warnings are true positives.

*Corresponding Author

ESEC/FSE '20, November 8–13, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-7043-1/20/11…$15.00
https://doi.org/10.1145/3368089.3409720

CCS CONCEPTS

• **Software and its engineering** → **Formal software verification**; • **Computing methodologies** → **Neural networks**.

KEYWORDS

Neural Network, Static Analysis, Numerical Bugs

ACM Reference Format:
Yuhao Zhang, Luyao Ren, Liqian Chen, Yingfei Xiong, Shing-Chi Cheung, and Tao Xie. 2020. Detecting Numerical Bugs in Neural Network Architectures. In *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '20), November 8–13, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3368089.3409720

1 INTRODUCTION

The use of deep neural networks (DNNs) within software systems (which are named as DL software systems) is increasingly popular, supporting critical classification tasks such as self-driving, facial recognition, and medical diagnosis. Construction of such systems requires training a DNN model based on a neural architecture scripted by a deep learning (DL) program[1]. To ease the development of DL programs, the developers popularly adopt various DL frameworks such as TensorFlow. A neural architecture, i.e., a network of tensors with a set of parameters, is captured by a computation graph configured to do one learning task. When these parameters are concretely bound after training based on the given training dataset, the architecture prescribes a DL model, which has been trained for a classification task.

To avoid unexpected or incorrect behaviors in DL software systems, it is necessary to detect bugs in their neural architectures. Although various approaches [8–10, 14, 17, 19–21, 23–25, 27–32] have been proposed to test or verify DL models, these approaches do not address the needs of two types of stakeholders: (1) architecture vendors who design and publish neural architectures to be

[1] A DL program may specify multiple neural architectures, each responsible for an assigned learning task. To ease the presentation, we assume that each DL program performs a single task with a neural architecture unless otherwise stated in this paper.

used by other users, and (2) developers who use neural architectures to train and deploy a model based on the developers' own training dataset.

- Architecture vendors need to provide quality assurance for their neural architecture. It is inadequate for the vendors to verify the architecture with specific instantiated models, which are dataset-dependent.
- Bugs in a neural architecture may manifest themselves into failures after developers have trained a model for hours, days, or even weeks, causing great loss in time and computation resources [34]. The loss can be prevented if these bugs can be detected early at the architecture level before model training.
- Failures can also occur when developers of a DL model need to retrain their models upon updates on training data. These updates can frequently happen during software system development and deployment, e.g., when the new feedback data is collected from users [33].
- Failures in DL models can be caused by a bug in the DL architecture, low-quality training data, incorrect parameter settings, or other issues. It is not easy for the developers to localize the bug.

In this paper, we present the first attempt to conduct static analysis for bug detection at the architecture level. Specifically, we target numerical bugs, an important category of bugs known to have catastrophic consequences. Numerical bugs are challenging to detect, often caused by complex component interactions and difficult to be spotted out during code review.

A neural architecture can contain numerical bugs that cause serious consequences. Numerical bugs in a neural architecture manifest themselves as numerical errors in the form of "NaN", "INF", or crashes during training or inference. For example, when a non-zero number is divided by zero, the result is "INF", indicating that it is an infinite number; when zero is divided by zero, the result is "NaN", indicating that it is not a number. When a numerical error occurs during training, the model trained using the buggy neural architecture becomes invalid. A numerical bug that manifests only when making inference is even more devastating: it can crash the software system or cause unexpected system behaviors when certain inputs are encountered during real system usage [24].

Detecting numerical bugs via testing is either too challenging at the architecture level or too late at the model level as revealed in previous empirical studies [15, 24, 34]. Testing an architecture is challenging as we cannot execute the architecture. Testing the trained models is too late to discover the bugs occurring at the training time as stated earlier.

To detect numerical bugs at the architecture level, in this paper, we propose to use static analysis because static analysis is able to cover the large combinatorial space imposed by the numerous parameters and possible inputs of a neural architecture. We propose a static analysis approach for detecting numerical bugs in neural architectures based on abstract interpretation [4], which mainly comprises two kinds of abstraction techniques, i.e., one for tensors and one for numerical values. We study three tensor abstraction techniques: array expansion, array smashing, and tensor partitioning, as well as two numerical abstraction techniques: interval abstraction and affine relation analysis. Among these techniques, array expansion, array smashing, and interval abstraction are adapted from existing abstraction techniques for imperative programs [1, 5]. In addition, to achieve scalability while maintaining adequate precision, we propose *tensor partitioning* to partition tensors and infer numerical information over partitions, based on our insight: many elements of a tensor are subject to the same operations. In particular, representing (concrete) tensor elements in a partition as one abstract element under appropriate abstract interpretation can reduce analysis effort by orders of magnitude. Motivated by this insight, *tensor partitioning* initially abstracts all elements in a tensor as one abstract element and iteratively splits each abstract element into smaller ones when its concrete elements go through different operations. Each abstract element represents one partition of the tensor, associated with a numerical interval that indicates the range of its concrete elements. Moreover, for the sake of precision, besides interval analysis, we conduct *affine relation analysis* to infer the elementwise affine equality relations among abstract elements representing partitions.

We evaluate the scalability and accuracy of our approach on two datasets: a set of 9 architectures with known numerical bugs collected by existing studies [24, 34], and a set of 48 large real-world neural architectures. In our evaluation, we design comparative experiments to study three tensor abstraction techniques and two numerical abstraction techniques. We specifically name the implementation of the combination scheme of tensor partitioning and affine relation analysis together with interval abstraction as DEBAR, being released as open source code[2]. In terms of scalability, the evaluation results show that array expansion is unscalable, and it times out in 33 architectures with a time budget of 30 minutes, while other techniques are scalable and can successfully analyze all architectures in 3 minutes. In terms of accuracy, DEBAR could achieve 93.0% accuracy with almost the same time performance compared to array smashing (87.1% accuracy) and (sole) interval abstraction (80.6% accuracy). These results demonstrate the effectiveness of tensor partitioning and affine relation analysis together with interval abstraction.

In summary, this paper makes three main contributions:

- A study of a static analysis approach for numerical bug detection in neural architectures, with three abstraction techniques for abstracting tensors and two for abstracting numerical values.
- Two abstraction techniques designed for analyzing neural architectures: tensor partitioning (for abstracting tensors) and (elementwise) affine relation analysis (for inferring numerical relations among tensor partitions).
- An evaluation on 9 buggy architectures in 48 real-world neural architectures, demonstrating the effectiveness of DEBAR.

2 OVERVIEW

In this section, we explain how our approach detects numerical bugs with an example in Listing 1. The example is modified from a

[2] https://github.com/ForeverZyh/DEBAR

real-world code snippet that creates rectangles and calculates the reciprocals of their areas[3].

```
1  # Input:
2  # center: 2*100-shape tensor whose elements in [-1, 1]
3  # offset: 2*100-shape tensor whose elements in [0, 2]
4
5  # Create 100 rectangles.
6  bottomLeft = center - offset
7  topRight = center + offset
8  rectangle = tf.concat([bottomLeft, topRight], axis=1)
9
10 # Calculate the reciprocal of their areas.
11 bottom, left, top, right = tf.split(rectangle,
       num_or_size_splits=4, axis=1)
12 width = right - left
13 height = top - bottom
14 area = width * height
15 scale = tf.reciprocal(area)
```

Listing 1: A Code Snippet of a Motivating Example

The program consists of two parts. The first part defines 100 rectangles, each by a central point and an offset vector. Input variable `center` represents 100 central points, where each point is a 2-element vector of 32-bit floats. Similarly, `offset` represents 100 offset vectors. Then from the central points and the offsets, the bottom left points and the top right points are calculated (Lines 6–7), and are concatenated into a 4*100 tensor to create rectangles (Line 8). The second part calculates the reciprocals of the areas. First, from `rectangle`, the bottom, left, top, and right coordinates are extracted (Line 11), each being a 1*100-shape tensor. The areas of rectangles are then calculated (Lines 12–14) followed by the calculation of their reciprocals (Line 15). This program contains a numerical bug that when any offset vector has an element of zero, the corresponding area becomes zero and the value of `scale` becomes `NaN`.

To capture the bug, an ideal way is to statically consider all possible values of `center` and `offset`, and check whether any of the values would result in a zero `area`. Abstract interpretation [4] is an effective solution to statically consider all possible values of variables. It analyzes the original program via an abstract domain, where each abstract value represents a set of concrete values. To apply abstract interpretation to our problem, the key is how to abstract a neural architecture. Given a neural architecture, we need to consider mainly two aspects. First, the numerical values and arithmetic computations need to be abstracted. Second, the tensors need to be abstracted. We first discuss three abstraction techniques adapted from existing work for analyzing imperative programs, and then describe two new techniques that we propose for analyzing neural architectures.

2.1 Interval Abstraction

Interval abstraction [5] is a popular abstract interpretation technique for abstracting numerical values, where each scalar variable v is represented by an interval, indicating the lower bound and the upper bound. These intervals are then calculated by mapping the standard operations into interval arithmetic. As a result, the following shows the calculations performed by the analysis.

[3] https://github.com/tensorflow/models/blob/13e7c85d521d7bb7cba0bf7d743366f7708b9df7/research/object_detection/box_coders/faster_rcnn_box_coder.py#L80

`center`:	all elements have $[-1, 1]$
`offset`:	all elements have $[0, 2]$
`bottomLeft`:	all elements have $[-3, 1]$
`topRight`:	all elements have $[-1, 3]$
`rectangle`:	the first two elements in each row have $[-3, 1]$, and the last two elements in each row have $[-1, 3]$
`bottom, left`:	all elements have $[-3, 1]$
`top, right`:	all elements have $[-1, 3]$
`width, height`:	all elements have $[-2, 6]$
`area`:	all elements have $[-12, 36]$

To detect numerical bugs, one can predefine the safe conditions for various operations, e.g., by restricting the argument not to take a zero value when calling `reciprocal`. Since zero is included in the interval of any element in `area`, a potential numerical bug is detected.

The first type of imprecision is introduced by interval abstraction. In the preceding example, we can conclude from the interval of `offset` that the elements in `area` are within interval $[0, 16]$, which is smaller than the inferred interval $[-12, 36]$. Since both `bottomLeft` and `topRight` are calculated from `center`, the effect of `center` is nullified when calculating `width` and `height`. However, such information is lost after the values have been abstracted into intervals. The imprecision may lead to false alarms. Consider a situation that the elements in `offset` contain values within interval $[1, 2]$, where no numerical error should be triggered. When analyzing using the interval abstraction, we would get $[-18, 36]$ for all elements in `area`, leading to a false alarm of numerical bugs.

2.2 Array Expansion

Array expansion [1] is a basic technique for abstracting an array in an imperative program to an abstract domain: the elements in the array are one-to-one mapped to the abstract domain, and no abstraction is performed. Mapping array expansion for tensor abstraction with interval abstraction, we can also directly map the elements in a tensor one-to-one to ranges in the abstract domain.

Scalability is the main problem of array expansion. The reason is that we need to record an interval for each element in a tensor, and in the motivating example, we need to record 200 intervals for `center` and 200 intervals for `offset`, substantially affecting scalability. As shown by our evaluation later, analyses using array expansion time out for most real-world models with a time budget of 30 minutes.

2.3 Array Smashing

Array smashing [1] is an alternative abstraction technique that uses one abstract element to represent all elements in a tensor. In this way, the number of abstract elements is greatly reduced. Mapping array smashing for tensor abstraction with interval abstraction, we use one range to cover all elements in a tensor.

`rectangle`:	$[-3, 3]$
`bottom, left, top, right`:	$[-3, 3]$
`width, height`:	$[-6, 6]$
`area`:	$[-36, 36]$

The second type of imprecision is introduced by array smashing. In the preceding example, the intervals of `center`, `offset`, `bottomLeft`, and `topRight` remain the same. Nevertheless, array

smashing can get the interval of `area` as $[-36, 36]$, which is less precise than what array expansion gets. In fact, when using array smashing, a warning would be reported for any input intervals as the difference between `bottomLeft` and `topRight` disappears when they are concatenated into `rectangle`.

2.4 Tensor Partitioning and Affine Relation Analysis

To scale up while maintaining adequate precision, we propose two techniques for abstracting tensors and numerical values, respectively. The first technique is *tensor partitioning*, which allows a tensor to be split into multiple partitions, where each partition is abstracted as a summary variable, and we maintain interval ranges for such summary variables. The second technique is *affine relation analysis*, which maintains affine relations between partitions and makes use of this relation to achieve more precise analysis.

Specifically, to support tensor partitioning, we maintain the set of indexes for each partition. We use $\mathbb{I}_A$ to denote the index ranges of the partition A, and use $*$ to denote all included indexes in a dimension. For example, the following shows the calculation process of the preceding example in tensor partitioning, where the names in uppercase represent partitions. Tensor `center` has one partition C including all its concrete elements. To support affine relation analysis, we introduce a symbolic summary variable a to denote each partition A (i.e., $\alpha(A) = a$), whose name is in lowercase. We use $\sigma(a)$ to denote its corresponding interval. With these summary variables, we maintain affine equality relations among these summary variables.

In the motivating example, the partition C corresponds to a symbolic summary variable c, whose interval range is $\sigma(c) = [-1, 1]$. Similarly, `offset` also has one partition O corresponding to an expression o, where the interval o is $[0, 2]$. Next, `bottomLeft` is calculated from `center` and `offset`. Since both `center` and `offset` have one partition, `bottomLeft` also has one partition BL, which corresponds to expression $c - o$ that is calculated from $\alpha(C) - \alpha(O)$. Following this process, we can calculate the partitions and maintain their affine equality relations.

1.	`center`:	$\mathbb{I}_C = * \times *,$	$\alpha(C) = c,$
2.		$\sigma(c) = [-1, 1]$	
3.	`offset`:	$\mathbb{I}_O = * \times *,$	$\alpha(O) = o,$
4.		$\sigma(o) = [0, 2]$	
5.	`bottomLeft`:	$\mathbb{I}_{BL} = * \times *,$	$\alpha(BL) = \alpha(C) - \alpha(O) = c - o$
6.	`topRight`:	$\mathbb{I}_{TR} = * \times *,$	$\alpha(TR) = \alpha(C) + \alpha(O) = c + o$
7.	`rectangle`:	$\mathbb{I}_{R_1} = [0..1] \times *,$	$\alpha(R_1) = \alpha(BL) = c - o$
8.		$\mathbb{I}_{R_2} = [2..3] \times *,$	$\alpha(R_2) = \alpha(TR) = c + o$
9.	`bottom`:	$\mathbb{I}_B = * \times *,$	$\alpha(B) = \alpha(R_1) = c - o$
10.	`left`:	$\mathbb{I}_L = * \times *,$	$\alpha(L) = \alpha(R_1) = c - o$
11.	`top`:	$\mathbb{I}_T = * \times *,$	$\alpha(T) = \alpha(R_2) = c + o$
12.	`right`:	$\mathbb{I}_R = * \times *,$	$\alpha(R) = \alpha(R_2) = c + o$
13.	`width`:	$\mathbb{I}_W = * \times *,$	$\alpha(W) = \alpha(R) - \alpha(L) = 2o$
14.	`height`:	$\mathbb{I}_R = * \times *,$	$\alpha(H) = \alpha(T) - \alpha(B) = 2o$
15.	`area`:	$\mathbb{I}_A = * \times *,$	$\alpha(A) = a,$
16.		$\sigma(a) = 2\sigma(o) \times 2\sigma(o) = [0, 16]$	

The calculation is dissimilar to interval abstraction with array smashing in two ways for the example. The first difference in calculation occurs at Lines 7 and 8. Since `rectangle` is concatenated from two tensors, we keep `rectangle` as two partitions, R_1 and R_2, each corresponding to an argument. In this way, we overcome the second type of imprecision brought by array smashing while keeping the number of abstract elements small. When splitting `rectangle` into `bottom`, `left`, `top`, and `width`, we can get the precise intervals for these tensors from the corresponding partitions in `rectangle`.

The second difference in calculation occurs at Line 13. When calculating *width*, we make use of the affine equality relations among partitions of R and L, and thus we can precisely infer that $width = 2o$ rather than only an imprecise interval for *width*. Similarly, the interval for `height` is also precise. In this way, we overcome the first type of imprecision due to interval abstraction. Finally, we calculate `area` based on `width` and `height`. Since the operation of $2o \times 2o$ is no longer linear, we cannot get any affine equality relations for `area`. Hence, we introduce a summary variable a for the whole `area`, and compute its interval range. Then, for `area` we get an interval $[0, 16]$, which is the ground-truth interval range of `area`.

3 APPROACH

In this section, we first introduce the preliminaries of abstract interpretation and two basic numerical abstract domains, the interval abstract domain, and the affine equality abstract domain. We then describe our abstraction for neural architectures using tensor partitioning (for abstracting tensors) and numerical abstractions, i.e., combining intervals with affine equalities (for abstracting numerical values). We then show how to abstract tensor operations under two abstraction techniques, tensor partitioning and affine relation analysis (as well as interval analysis), designed for neural architectures. We also discuss the initial intervals for input ranges and parameter ranges.

3.1 Preliminaries

3.1.1 Basics of Abstract Interpretation. In abstract interpretation [4], concrete properties are described in the concrete domain $\mathbb{C}$ with a partial order $\subseteq$, and abstract properties are described in the abstract domain $\mathbb{A}$ with a partial order $\sqsubseteq$. We say that the correspondence between concrete properties and abstract properties is a Galois connection $\langle \mathbb{C}, \subseteq \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle \mathbb{A}, \sqsubseteq \rangle$ with an abstraction function $\alpha : \mathbb{C} \mapsto \mathbb{A}$ and a concretization function $\gamma : \mathbb{A} \mapsto \mathbb{C}$ satisfying

$$\forall c \in \mathbb{C}, \forall a \in \mathbb{A}. \alpha(c) \sqsubseteq a \Leftrightarrow c \subseteq \gamma(a).$$

To infer the value range for variables in a DL program, we need to compute the possible sets of values that each variable can take. We define the concrete domain $\mathbb{C}$ of n variables as $\mathcal{P}(\mathbb{R}^n)$, where an element is a set of n-element vectors denoting the possible values that n variables can take. The partial order in $\mathbb{C}$ is the subset relation $\subseteq$ over sets.

3.1.2 Abstract Domain of Intervals. The abstract domain of intervals [3] $\mathbb{A}_\mathrm{I}$ is defined as

$$\mathbb{A}_\mathrm{I} \triangleq \{([l_1, u_1], \ldots, [l_n, u_n]) \mid l, u \in \mathbb{R}^n\}.$$

An element in $\mathbb{A}_\mathrm{I}$ can be seen as a pair of two vectors (l, u), where $[l_i, u_i]$ denotes the lower bound and upper bound of the values that the i-th variable may take. Given two elements $a_1, a_2 \in \mathbb{A}_\mathrm{I}$,

we say $a_1 \sqsubseteq a_2$ if both have n intervals and each interval in a_1 is a sub-interval of the interval in the corresponding position in a_2.

The abstraction function α_{I} of an element $c \in \mathbb{C}$ is defined as $\alpha_{\mathrm{I}}(c) = ([l_1^c, u_1^c], \ldots, [l_n^c, u_n^c])$, where

$$l_i^c = \min_{x \in c}(x_i), \quad u_i^c = \max_{x \in c}(x_i), \qquad 1 \le i \le n.$$

The concretization function γ_{I} of an element $a \in \mathbb{A}_{\mathrm{I}}$ is defined as

$$\gamma_{\mathrm{I}}(a) = \{x \in \mathbb{R}^n \mid \forall i \in [1, n].x_i \in [l_i^a, u_i^a]\},$$

where $[l_i^a, u_i^a]$ is the interval range of the i-th variable of a.

It is easy to see that the concrete domain $\langle \mathbb{C}, \subseteq \rangle$ and the interval abstract domain $\langle \mathbb{A}_{\mathrm{I}}, \sqsubseteq_{\mathrm{I}} \rangle$ form the Galois connection. More details can be found in the publication by Cousot and Cousot [3].

3.1.3 Abstract Domain of Affine Equalities. As discussed in Section 2, we also maintain affine relations among variables in a DL program in the form of

$$\sum_{i>0} \omega_i x_i = \omega_0, \tag{1}$$

where x_i's are variables and ω_i's are constant coefficients, inferred automatically during the analysis.

The abstract domain of affine equalities [16] $\mathbb{A}_{\mathrm{E}}$ is defined as

$$\mathbb{A}_{\mathrm{E}} \triangleq \{(\mathbf{A}, b) \mid \mathbf{A} \in \mathbb{R}^{m \times n}, b \in \mathbb{R}^m, m > 0\},$$

where a matrix $\mathbf{A}$ and a column vector b define the affine space of n variables. An element in $\mathbb{A}_{\mathrm{E}}$ constrains variables $x \in \mathbb{R}^n$ by an equation $\mathbf{A}x = b$ describing the possible set of values that x can take. Furthermore, to have a canonical form, we require $(\mathbf{A}, b)$ to be in the reduced row echelon form [16].

The abstraction function α_{E} of an element $c \in \mathbb{C}$ is defined as

$$\alpha_{\mathrm{E}}(c) = \begin{cases} (\mathbf{A}, b), & (\mathbf{A}, b) \text{ is in reduced row echelon form, and} \\ & \mathbf{A}x = b \text{ holds for all } x \text{ in } c \\ \top, & \text{if } c \text{ is the whole space} \\ \bot, & \text{otherwise.} \end{cases}$$

The concretization function γ_{E} of an element $a^{\sharp \mathrm{E}} = (\mathbf{A}, b) \in \mathbb{A}_{\mathrm{E}}$ is defined as

$$\gamma_{\mathrm{E}}((\mathbf{A}, b)) = \{x \in \mathbb{R}^n \mid \mathbf{A}x = b\}.$$

The concrete domain $\langle \mathbb{C}, \subseteq \rangle$ and the affine equality abstract domain $\langle \mathbb{A}_{\mathrm{E}}, \sqsubseteq_{\mathrm{E}} \rangle$ form the Galois connection

$$\langle \mathbb{C}, \subseteq \rangle \underset{\alpha_{\mathrm{E}}}{\overset{\gamma_{\mathrm{E}}}{\leftrightarrows}} \langle \mathbb{A}_{\mathrm{E}}, \sqsubseteq_{\mathrm{E}} \rangle.$$

The details about the domain operations (including meet, join, inclusion test, etc.) of the affine equality abstract domain can be found in the publication by Karr [16]. We do not need a widening operation for the domain of affine equalities because the lattice of affine equalities has finite height, and the number of affine equalities while analyzing a program is decreasing until reaching the dimension of the affine space in the program.

3.2 Abstraction for Neural Architectures

We use tensor partitioning and interval abstraction with affine equality relation to abstract tensors in neural architectures.

3.2.1 Tensor Partitioning. As discussed in Section 2, we introduce a new granularity of array abstraction, named **tensor partitioning**, which is a form of array partitioning (also named array segmentation) [6, 12, 13] but tailored for tensor operations: we partition a tensor A into a set of disjoint partitions $\{A_1, A_2, \ldots, A_n\}$ where each partition A_i is a sub-tensor of A. The number of partitions of A is denoted as $\mathcal{N}_A$. The set of array indexes of the cells from partition A_i is continuous in A and defined by Cartesian products of index intervals for all dimensions, denoted as $\mathbb{I}_{A_i}$. Note that the indexes in $\mathbb{I}_{A_i}$ are indexes of the corresponding elements in tensor A, while we sometimes use $\mathbb{I}_{A_i.shape}$ to represent the indexes of the corresponding elements in sub-tensor A_i, where $A_i.shape$ denotes a tuple of integers giving the size of the sub-tensor A_i along each dimension. In our motivating example, R_2 is a partition of `rectangle`, and $\mathbb{I}_{R_2} = [2..3] \times [0..99]$, $R_2.shape = (2, 100)$, $\mathbb{I}_{R_2.shape} = [0..1] \times [0..99]$. For clarity, we introduce a notion of *partitioning positions* for each dimension to denote the indexes where we partition the tensor in that dimension. Index i is a partitioning position for a tensor A in dimension p iff the element $A[i]$ and the element $A[i+1]$ in dimension p belong to different partitions. It is worth mentioning that the partitioning positions are easier to infer for DNN implementations than for regular programs (e.g., C programs) [6, 12, 13], because the shapes of (sub-)tensors are usually determined syntactically (often specified by parameters of tensor operations) so that we know the exact boundary of each partition, e.g., *tf.concat* and *tf.split* in our motivating example.

After partitioning, for each partition A_i, we introduce an abstract summary variable a_i to subsume all the elements in A_i, denoted as $\alpha(A_i) = a_i$. Note that in this paper, we use lowercase letters to denote the summary variables of partitions (sub-tensors) while uppercase letters to denote tensors. To perform static analysis, we maintain numerical relations among summary variables of partitions, with details described in the next subsection.

3.2.2 Interval Abstraction with Affine Equality Relation. We combine the interval abstraction and affine equality relation abstraction as our numerical abstraction to infer the value range for scalar variables in the DL programs and also for those auxiliary abstract summary variables introduced by tensor partitioning. We could use relational numerical abstract domains (such as polyhedra) to infer (inequality) relations. However, because many tensor operations induce affine equality relations, in this paper, we consider only the affine equality relations among variables. In addition, affine equality relations are cheap to infer, and thus are fit to analyze large DNNs.

Furthermore, because *ReLU* operations are widely used in DNN implementations, for each variable a, we introduce a^{ReLU} to denote the resulting variable of $ReLU(a)$, i.e., $a^{ReLU} = max(0, a)$. Considering the way of using *ReLU* operations in DNN implementations, in this paper, we maintain only the affine equality relations between a variable b and the *ReLU* result of another variable a of the same shape (while a, b may be the summary variables of partitions from different tensors), in the form of

$$b - a^{ReLU} = 0,$$

which can also be expressed in the form of Eq 1. Additionally, the *ReLU* operation has a property

$$ReLU(a) - ReLU(-a) - a = 0, \forall a \in \mathbb{R}.$$

We can utilize this property for better analysis precision by (1) adding an additional equality

$$a^{ReLU} - a^{-ReLU} - a = 0,$$

where a^{-ReLU} denotes the result of $ReLU(-a)$; (2) adding an additional equality

$$c^{ReLU} - a^{-ReLU} = 0$$

for every equality in the form of $c = -a$.

3.2.3 *Abstract Domain for Neural Architectures.*

Definition 3.1. *The abstract domain for **T**ensor partitioning and **I**nterval abstraction with affine **E**quality relation $\mathbb{A}_{\text{TIE}}$ is defined as*

$$\mathbb{A}_{\text{TIE}} \triangleq \{(\mathcal{P}, a^{\sharp \text{I}}, a^{\sharp \text{E}}) \mid a^{\sharp \text{I}} \in \mathbb{A}_{\text{I}}, a^{\sharp \text{E}} \in \mathbb{A}_{\text{E}}\},$$

where $\mathcal{P} = \{A_1, \ldots, A_n\}$ is the set of the disjoint partitions of the tensors and $a^{\sharp \text{I}}, a^{\sharp \text{E}}$ are the numerical abstract elements over the n summary variables corresponding to the partitions $\mathcal{P}$ in the interval domain $\mathbb{A}_{\text{I}}$ and the affine equality domain $\mathbb{A}_{\text{E}}$, respectively.

Definition 3.2. *The concretization function γ_{TIE} of an element $a^\sharp = (\mathcal{P}, a^{\sharp \text{I}}, a^{\sharp \text{E}}) \in \mathbb{A}_{\text{TIE}}$ is defined as*

$$\gamma_{\text{TIE}}(a^\sharp) = \left\{ A \;\middle|\; \begin{array}{c} \mathcal{P} = \{A_1, \ldots, A_n\} \wedge \forall (j_1, \ldots, j_n) \in J. \\ (A_1[j_1], \ldots, A_n[j_n]) \in (\gamma_{\text{I}}(a^{\sharp \text{I}}) \cap \gamma_{\text{E}}(a^{\sharp \text{E}})) \end{array} \right\},$$

where A is the tensor constructed by its partitions $\mathcal{P} = \{A_1, \ldots, A_n\}$ and

$$J = \{(j_1, \ldots, j_n) \mid \forall i \in [1, n].j_i \in \mathbb{I}_{A_i.shape}\}.$$

3.3 Abstracting Tensor Operations

We next show how to construct abstract operations based on tensor partitioning and affine relation analysis (together with interval analysis) for analyzing three common tensor operations in neural architectures. We provide the construction for other operations in our DEBAR open source code.

To ease the presentation, we illustrate our approach using vectors (one-dimensional tensors) and matrices (two-dimensional tensors). Our approach is generalizable to multi-dimensional tensors.

3.3.1 *Addition and Subtraction.*

Tensor addition and subtraction, in the form of $C = A \pm B$, take two input tensors A, B with the same shape to calculate the resulting tensor C.

Since the input tensors A and B may not be partitioned in the same way, we should first align the partitions of A and those of B, such that $\forall i \in [1, \mathcal{N}_A].\mathbb{I}_{A_i} = \mathbb{I}_{B_i}$ where $\mathcal{N}_A$ denotes the number of partitions of A and B after aligned. To align the partitions of A and those of B, for each dimension, we take the set union of the two partitioning positions as the new set of partitioning positions for A and B. Then, we put the aligned set of partitioning positions as that for the resulting tensor C, such that C is aligned with A, B.

After that, for each partition C_i, we compute the interval range for its summary variable by

$$\sigma(c_i) = \sigma(a_i) \pm \sigma(b_i).$$

Furthermore, for tensor addition and subtraction, we maintain the elementwise affine equality relations among partitions of A, B, C. For each partition C_i, we have

$$c_i = a_i \pm b_i \qquad i \in \{1, \ldots, \mathcal{N}_C\},$$

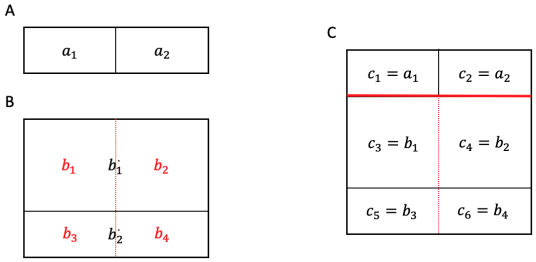

Figure 1: Concatenating Tensors (Horizontally)

which means

$$\forall j \in \mathbb{I}_{(C_i.shape)}.C_i[j] = A_i[j] \pm B_i[j] \quad i \in \{1, \ldots, \mathcal{N}_C\}.$$

For example, suppose that both the one-dimensional tensors A and B are partitioned into two partitions, and

$$\begin{array}{c} \mathbb{I}_{A_1} = [0..2] \wedge \mathbb{I}_{A_2} = [3..9] \wedge \mathbb{I}_{B_1} = [0..5] \wedge \mathbb{I}_{B_2} = [6..9] \wedge \\ \alpha(A_1) = a_1 \wedge \sigma(a_1) = 1 \wedge \alpha(A_2) = a_2 \wedge \sigma(a_2) = [2, 3] \wedge \\ \alpha(B_1) = b_1 \wedge \sigma(b_1) = 4 \wedge \alpha(B_2) = b_2 \wedge \sigma(b_2) = [5, 6]. \end{array}$$

After aligning partitions of A and B, we have

$$\begin{array}{c} \mathbb{I}_{A_1} = [0..2] \wedge \mathbb{I}_{A_2} = [3..5] \wedge \mathbb{I}_{A_3} = [6..9] \wedge \\ \mathbb{I}_{B_1} = [0..2] \wedge \mathbb{I}_{B_2} = [3..5] \wedge \mathbb{I}_{B_3} = [6..9] \wedge \\ \alpha(A_1) = a_1 \wedge \sigma(a_1) = 1 \wedge \alpha(A_2) = a_2 \wedge \sigma(a_2) = [2, 3] \wedge \\ \alpha(A_3) = a_3 \wedge \sigma(a_3) = [2, 3] \wedge \alpha(B_1) = b_1 \wedge \sigma(b_1) = 4 \wedge \\ \alpha(B_2) = b_2 \wedge \sigma(b_2) = 4 \wedge \alpha(B_3) = b_3 \wedge \sigma(b_3) = [5, 6]. \end{array}$$

After $C = A \pm B$, for C, we have:

$$\begin{array}{c} \mathbb{I}_{C_1} = [0..2] \wedge \mathbb{I}_{C_2} = [3..5] \wedge \mathbb{I}_{C_3} = [6..9] \wedge \\ \alpha(C_1) = c_1 \wedge \sigma(c_1) = 1 \pm 4 \wedge \alpha(C_2) = c_2 \wedge \sigma(c_2) = [2, 3] \pm 4 \\ \wedge \alpha(C_3) = c_3 \wedge \sigma(c_3) = [2, 3] \pm [5, 6] \wedge \\ c_i = a_i \pm b_i \quad i = \{1, 2, 3\}. \end{array}$$

3.3.2 *Concatenate.*

In DNN implementations, tensors can be concatenated along certain dimension p. An assignment statement $C = Concatenate(A, B, p)$ denotes that two input tensors A and B are concatenated to form an output tensor C along dimension p. For example, Figure 1 shows that two two-dimensional tensors A and B are concatenated to form a tensor C along dimension 0 (rows). To handle the *Concatenate* operation, first, we need to align partitions of A and B along all other dimensions except dimension p. To align the partitions, in each dimension (except dimension p), we use the set union of partitioning positions of A and B as the new set of partitioning positions for A, B and also that for C. In dimension p, we do not change the partitioning positions of A and B, while the set of partitioning positions of C consists of (1) A's partitioning positions in dimension p; (2) the size of A in dimension p, denoted as n (representing the boundary between A and B); (3) B's partitioning positions (in dimension p) plus n. Let $\mathcal{N}_A, \mathcal{N}_B$ denote the number of partitions of A, B after alignment, respectively.

For simplicity of presentation, here we assume that two-dimensional tensors A, B are concatenated to form a tensor C along dimension 0 (i.e., $p = 0$). Then for each partition C_i, we calculate its interval range by

$$\sigma(c_i) = \begin{cases} \sigma(a_i) & \text{if } i \in \{1, \ldots, \mathcal{N}_A\} \\ \sigma(b_{i-\mathcal{N}_A}) & \text{if } i \in \{\mathcal{N}_A + 1, \ldots, \mathcal{N}_A + \mathcal{N}_B\}. \end{cases}$$

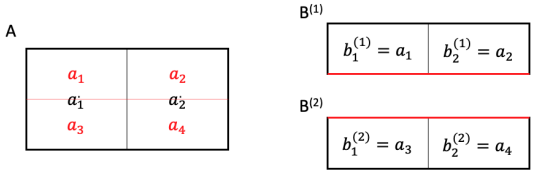

Figure 2: Splitting Tensors (Vertically)

We also maintain the elementwise affine equality relations between C_i and A_i (when $i \leq \mathcal{N}_A$), as well as relations between C_i and B_i (when $i > \mathcal{N}_A$), as

$$c_i = a_i \qquad i \in \{1, \ldots, \mathcal{N}_A\}$$
$$c_i = b_{i-\mathcal{N}_A} \qquad i \in \{\mathcal{N}_A + 1, \ldots, \mathcal{N}_A + \mathcal{N}_B\},$$

which means

$$\forall (j,k) \in \mathbb{I}_{(C_i.shape)}.C_i[j][k] = A_i[j][k] \quad 1 \leq i \leq \mathcal{N}_A$$
$$\forall (j,k) \in \mathbb{I}_{(C_i.shape)}.C_i[j][k] = B_{i-\mathcal{N}_A}[j][k] \quad \mathcal{N}_A < i \leq \mathcal{N}_A + \mathcal{N}_B.$$

For the example shown in Figure 1, suppose

$$\mathbb{I}_{A_1} = [0..0] \times [0..1] \wedge \mathbb{I}_{A_2} = [0..0] \times [2..3] \wedge$$
$$\mathbb{I}_{B_1} = [0..1] \times [0..3] \wedge \mathbb{I}_{B_2} = [2..2] \times [0..3] \wedge$$
$$\alpha(A_1) = a_1 \wedge \alpha(A_2) = a_2 \wedge \alpha(B_1) = b_1^{\cdot} \wedge \alpha(B_2) = b_2^{\cdot},$$

where we temporarily use $b_i^{\cdot}$ to denote the summary variables for B_i here. After aligning the partitions of A and B , we have

$$\mathbb{I}_{A_1} = [0..0] \times [0..1] \wedge \mathbb{I}_{A_2} = [0..0] \times [2..3] \wedge$$
$$\mathbb{I}_{B_1} = [0..1] \times [0..1] \wedge \mathbb{I}_{B_2} = [0..1] \times [2..3] \wedge$$
$$\mathbb{I}_{B_3} = [2..2] \times [0..1] \wedge \mathbb{I}_{B_4} = [2..2] \times [2..3] \wedge$$
$$\alpha(A_1) = a_1 \wedge \alpha(A_2) = a_2 \wedge$$
$$\alpha(B_1) = b_1 \wedge \sigma(b_1) = \sigma(b_1^{\cdot}) \wedge \alpha(B_2) = b_2 \wedge \sigma(b_2) = \sigma(b_1^{\cdot}) \wedge$$
$$\alpha(B_3) = b_3 \wedge \sigma(b_3) = \sigma(b_2^{\cdot}) \wedge \alpha(B_4) = b_4 \wedge \sigma(b_4) = \sigma(b_2^{\cdot}).$$

Then, after $C = Concatenate(A, B, 0)$, for C, we have:

$$\mathbb{I}_{C_1} = [0..0] \times [0..1] \wedge \mathbb{I}_{C_2} = [0..0] \times [2..3] \wedge \mathbb{I}_{C_3} = [1..2] \times [0..1] \wedge$$
$$\mathbb{I}_{C_4} = [1..2] \times [2..3] \wedge \mathbb{I}_{C_5} = [3..3] \times [0..1] \wedge \mathbb{I}_{C_6} = [3..3] \times [2..3] \wedge$$
$$\alpha(C_1) = c_1 \wedge \sigma(c_1) = \sigma(a_1) \wedge \alpha(C_2) = c_2 \wedge \sigma(c_2) = \sigma(a_2) \wedge$$
$$\alpha(C_3) = c_3 \wedge \sigma(c_3) = \sigma(b_1) \wedge \alpha(C_4) = c_4 \wedge \sigma(c_4) = \sigma(b_2) \wedge$$
$$\alpha(C_5) = c_5 \wedge \sigma(c_5) = \sigma(b_3) \wedge \alpha(C_6) = c_6 \wedge \sigma(c_6) = \sigma(b_4) \wedge$$
$$c_i = a_i \qquad i = \{1,2\} \wedge c_i = b_{i-2} \qquad i = \{3,4,5,6\}.$$

3.3.3 Split. A tensor can be split into sub-tensors along a certain dimension. More clearly, a statement $(B^{(1)}, \ldots, B^{(n)}) = split(A, n, p)$ denotes that A is split along dimension p into n smaller tensors, which are stored in $B^{(1)}, \ldots, B^{(n)}$. The statement requires that n evenly divides $A.shape[p]$ (i.e., the number of elements in dimension p in A). For example, Figure 2 shows that a two-dimensional tensor A is split along dimension 0 (rows) into 2 sub-tensors B_1 and B_2. To handle the *Split* operation, first, we use the following set as the new set of partitioning positions of A in dimension p: $\{\frac{A.shape[p]}{n} - 1, 2 * \frac{A.shape[p]}{n} - 1, \ldots, (n-1) * \frac{A.shape[p]}{n} - 1\}$, and align the partitions of A with respect to the new set of partitioning positions. Let $\mathcal{N}_A$ denote the number of partitions of A after alignment. Then, we keep the set of partitioning positions of A as that of $B^{(j)}$ for all dimensions except p. In dimension p, we use the empty set as the set of partitioning positions of $B^{(j)}$. In other words, we do not partition $B^{(j)}$ in dimension p.

For simplicity of presentation, here we assume that two-dimensional tensors A are split into sub-tensors $(B^{(1)}, \ldots, B^{(n)})$ along dimension 0 (i.e., $p = 0$). Then, considering an output sub-tensor $B^{(j)}$, for each of its partitions $B_i^{(j)}$, we calculate its interval range by

$$\sigma(b_i^{(j)}) = \sigma(a_{i'}),$$

where $i' = (j-1) * \frac{\mathcal{N}_A}{n} + i$. We also maintain the elementwise affine equality relations between $B_i^{(j)}$ and $A_{i'}$ (where $i' = (j-1) * \frac{\mathcal{N}_A}{n} + i$):

$$b_i^{(j)} = a_{i'} \quad i \in \{1, \ldots, \frac{\mathcal{N}_A}{n}\},$$

which means

$$\forall (k,m) \in (B_i^{(j)}.shape).B_i^{(j)}[k][m] = A_{i'}[k][m] \quad i \in \{1, \ldots, \frac{\mathcal{N}_A}{n}\}.$$

For example, consider $(B^{(1)}, B^{(2)}) = split(A, 2, 0)$ in Figure 2 and suppose that before this statement, A is partitioned into the following two partitions:

$$\mathbb{I}_{A_1} = [0..1] \times [0..1] \wedge \mathbb{I}_{A_2} = [0..1] \times [2..3] \wedge \alpha(A_1) = a_1^{\cdot} \wedge \alpha(A_2) = a_2^{\cdot},$$

where we temporarily use $a_i^{\cdot}$ to denote the summary variables for A_i here. After aligning the partitions of A, we have

$$\mathbb{I}_{A_1} = [0..0] \times [0..1] \wedge \mathbb{I}_{A_2} = [0..0] \times [2..3] \wedge$$
$$\mathbb{I}_{A_3} = [1..1] \times [0..1] \wedge \mathbb{I}_{A_4} = [1..1] \times [2..3] \wedge$$
$$\alpha(A_1) = a_1 \wedge \sigma(a_1) = \sigma(a_1^{\cdot}) \wedge \alpha(A_2) = a_2 \wedge \sigma(a_2) = \sigma(a_2^{\cdot}) \wedge$$
$$\alpha(A_3) = a_3 \wedge \sigma(a_3) = \sigma(a_1^{\cdot}) \wedge \alpha(A_4) = a_4 \wedge \sigma(a_4) = \sigma(a_2^{\cdot}).$$

Then, after $(B^{(1)}, B^{(2)}) = split(A, 2, 0)$, for $B^{(1)}, B^{(2)}$, we have:

$$\mathbb{I}_{B_1^{(1)}} = [0..0] \times [0..1] \wedge \mathbb{I}_{B_2^{(1)}} = [0..0] \times [2..3] \wedge$$
$$\mathbb{I}_{B_1^{(2)}} = [0..0] \times [0..1] \wedge \mathbb{I}_{B_2^{(2)}} = [0..0] \times [2..3] \wedge$$
$$\alpha(B_1^{(1)}) = b_1^{(1)} \wedge \sigma(b_1^{(1)}) = \sigma(a_1) \wedge \alpha(B_2^{(1)}) = b_2^{(1)} \wedge \sigma(b_2^{(1)}) = \sigma(a_2) \wedge$$
$$\alpha(B_1^{(2)}) = b_1^{(2)} \wedge \sigma(b_1^{(2)}) = \sigma(a_3) \wedge \alpha(B_2^{(2)}) = b_2^{(2)} \wedge \sigma(b_2^{(2)}) = \sigma(a_4) \wedge$$
$$b_i^{(j)} = a_{i'} \quad i, j = \{1,2\},$$

where $i' = (j-1) * 2 + i$.

3.4 Input Ranges and Parameter Ranges

In previous sections, we initialize the intervals as full ranges of the respective types, e.g., `[FLOAT_MIN, FLOAT_MAX]` for floats. However, in the real world, the input may fall into only a small range. For example, an RGB value of an image falls into $[0, 255]$. Especially, in many applications, inputs are normalized into a small range (e.g., $[-1, 1]$) after the preprocessing step. Similarly, the parameters of a neural network may also fall into a small range. For example, many neural architectures are initialized with a weight initialization function, which reflects the desired upper bounds and lower bounds of the parameters. Assuming full ranges for these inputs and parameters may lead to unnecessary false positives, and thus our approach also allows the user to specify input ranges and parameter ranges, and uses the user-provided ranges to initialize the intervals.

4 EVALUATION

Our evaluation aims to answer the following research questions for assessing effectiveness of DEBAR (RQ1) and studying the techniques in DEBAR (RQ2 and RQ3):

- RQ1: Is DEBAR effective in detecting numerical bugs?
- RQ2: How effective are the three tensor abstraction techniques?
- RQ3: How effective are the two numerical abstraction techniques?

4.1 Datasets

We collect two datasets for the evaluation. The first dataset is a set of 9 buggy architectures collected by existing studies. The buggy architectures come from two studies: 8 architectures were collected by a previous empirical study [34] on TensorFlow bugs and 1 architecture was obtained from a study conducted to evaluate TensorFuzz [24].

As most of the architectures in the first dataset are small, we collect the second dataset, which contains 48 architectures from a large collection of research projects in the repository of TensorFlow Research Models[4]. The whole collection contains 66 projects implemented in TensorFlow by researchers and developers for different tasks in various domains, including computer vision, natural language processing, speech recognition, and adversarial machine learning. We first filter out the projects that are not related to specific neural architectures such as API frameworks and optimizers. We further filter out the projects of which the computation graph cannot be generated due to incomplete documentation or complicated configuration. As a result, 32 projects remain after filtering, and some of them contain more than one neural architecture. Overall, our second dataset contains a great diversity of neural architectures such as Convolutional Neural Network (CNN), Recurrent Neural Network (RNN), Generative Adversarial Network (GAN), and Hidden Markov Model (HMM). Note that we have no knowledge about whether the architectures in this dataset contain numerical bugs when collecting the dataset.

For every architecture in the two datasets, we extract the computation graph via a TensorFlow API. Each extracted computation graph is represented by a Protocol Buffer file[5], which provides the operations (nodes) and the data flow relations (edges). We make 48 computation graphs publicly available[6].

Columns 1–4 in Table 1 provide an overview of the two datasets. Column 2 provides an estimation of the lines of code in the corresponding DL programs. Column 3 shows the number of operations in the computation graphs, and *textsum* has the highest number of operations (208,412). Moreover, Column 4 shows the number of parameters (trainable weights) in the DNN architectures, and *lm_1b* has the largest number of parameters (1.04G).

4.2 Setups of Input Range and Parameter Range

In our evaluation, we conservatively provide the input ranges. As described in Section 3.4, we get the initial input ranges from the physical meaning of inputs, and derive input range information from the preprocessing programs typically written for the training data. If we fail to provide input ranges with the preceding two steps, we set the input ranges to `[FLOAT_MIN, FLOAT_MAX]`.

We determine the parameter ranges with the weight initialization functions. If the parameters are initialized to zero, we set their ranges as default values $[-1, 1]$. We also provide some heuristics for uninitialized parameters: setting *variance* to `[0, FLOAT_MAX]`, and setting *count* and *step* to `[1, FLOAT_MAX]`. Otherwise, we set the parameter ranges to `[FLOAT_MIN, FLOAT_MAX]`.

We provide the setups for each architecture in our DEBAR open source code.

4.3 Unsafe Operations to Check

By investigating the dataset in a previous empirical study [34], we collect a list of unsafe operations shown in Table 2. These operations have the most frequent occurrences and have a high potential to cause numerical errors. In this paper, we use our static analysis approach based on abstract interpretation to check whether these operations can cause numerical errors. Specifically, after performing our analysis, we can get the interval range for the parameter x of the operations, denoted as $[\underline{x}, \overline{x}]$. Then we check the unsafe constraints listed in Table 2. If the unsafe constraints for an operation are satisfiable, our checker issues an alarm for indicating that the operation may cause numerical errors. Otherwise, the operation is safe. In Table 2, Mf and mf, respectively, denote the largest non-infinity floating-point number and the smallest non-zero positive floating-point number that the used floating-point format (e.g., 32-bit, 64-bit) can represent exactly.

In DNN implementations, there are many operations (e.g., the multiplication) that may lead to numerical bugs. Our approach's current implementation checks only those operations listed in Table 2, but can be easily extended to other operations.

4.4 Measurements

Our approach checks every operation that may lead to a numerical error and determines whether a warning should be reported. To measure the effectiveness of our approach, we treat it as a classifier that classifies whether each operation is buggy, and evaluates its effectiveness using the number of true/false positives/negatives and accuracy. More concretely, true (false) positives refer to the warnings that are (not) indeed bugs, true (false) negatives refer to those correct (buggy) operations where no warning is reported, and accuracy is calculated using the following formula, where TP/FP refers to true/false positive and TN/FN refers to true/false negative.

$$Accuracy = \frac{\#TP + \#TN}{\#TP + \#TN + \#FP + \#FN}$$

For the first dataset, we refer to user patches to determine whether a warning is a bug. For the second dataset,

- 204 true positives are confirmed by executing the architecture under analysis using the designed inputs and parameters to trigger the numerical errors.
- 52 true positives are confirmed by the developer-provided fixes (not merged yet) in the issue discussion.
- 43 true positives are confirmed when two authors of this paper separately do reasoning on each computation graph, and both authors conclude that each warning is true positive.

[4] https://github.com/tensorflow/models/blob/13e7c85d521d7bb7cba0bf7d743366f7708b9df7/research
[5] https://en.wikipedia.org/wiki/Protocol_Buffers
[6] https://doi.org/10.5281/zenodo.3843648

Table 1: Dataset Overview and Results

Name	LoC	#Ops	#Params	TP	DEBAR				Array Smashing				Sole Interval Abstraction			
					TN	FP	Acc	Time	TN	FP	Acc	Time	TN	FP	Acc	Time
TensorFuzz	77	225	178K	4	0	0	100.0%	1.9	0	0	100.0%	1.9	0	0	100.0%	1.7
Github-IPS-1	367	1,546	5.05M	1	4	0	100.0%	2.3	4	0	100.0%	2.2	4	0	100.0%	2.2
Github-IPS-6	2,377	167	23.6K	2	0	0	100.0%	1.7	0	0	100.0%	1.7	0	0	100.0%	1.7
Github-IPS-9	226	102	23.6K	1	0	0	100.0%	1.7	0	0	100.0%	1.7	0	0	100.0%	1.7
StackOverflow-IPS-1	102	329	9.28M	1	1	0	100.0%	1.8	1	0	100.0%	1.8	1	0	100.0%	1.8
StackOverflow-IPS-2	102	329	9.28M	1	1	0	100.0%	1.8	1	0	100.0%	1.7	1	0	100.0%	1.8
StackOverflow-IPS-6	102	329	9.28M	1	1	0	100.0%	1.8	1	0	100.0%	1.8	1	0	100.0%	1.8
StackOverflow-IPS-7	49	145	407K	2	0	0	100.0%	1.7	0	0	100.0%	1.7	0	0	100.0%	1.7
StackOverflow-IPS-14	48	74	7.85K	1	0	0	100.0%	1.7	0	0	100.0%	1.7	0	0	100.0%	1.6
ssd_mobile_net_v1	71,242	22,412	27.3M	26	233	48	84.4%	21.8	137	144	53.1%	21.5	136	145	52.8%	21.6
ssd_inception_v2	71,242	23,929	100M	3	49	2	96.3%	19.8	45	6	88.9%	19.8	44	7	87.0%	19.7
ssd_mobile_net_v2	71,242	28,724	24.3M	26	233	48	84.4%	25.5	137	144	53.1%	26.0	136	145	52.8%	25.9
faster_rcnn_resnet_50	71,242	12,485	73.4M	5	31	22	62.1%	11.4	31	22	62.1%	11.4	31	22	62.1%	11.5
deep_speech	659	7,318	0.131K	0	6	0	100.0%	6.9	6	0	100.0%	7.1	6	0	100.0%	7.1
deeplab	7,514	21,100	87.1M	0	2	0	100.0%	17.8	2	0	100.0%	17.7	2	0	100.0%	18.3
autoencoder_mnae	369	187	944K	0	1	0	100.0%	2.6	1	0	100.0%	2.7	1	0	100.0%	2.6
autoencoder_vae	369	370	1.41M	2	1	0	100.0%	2.7	1	0	100.0%	2.7	1	0	100.0%	2.7
attention_ocr	1,772	3,624	1.74M	1	4	2	71.4%	5.2	4	2	71.4%	5.1	4	2	71.4%	5.2
textsum	906	208,412	10.5M	0	94	0	100.0%	105.7	94	0	100.0%	103.1	94	0	100.0%	106.4
shake_shake_32	1,233	7,348	5.85M	0	55	0	100.0%	7.5	55	0	100.0%	7.6	55	0	100.0%	7.7
shake_shake_96	1,233	7,348	52.4M	0	55	0	100.0%	7.6	55	0	100.0%	7.7	55	0	100.0%	7.7
shake_shake_112	1,233	7,348	71.3M	0	55	0	100.0%	7.6	55	0	100.0%	7.6	55	0	100.0%	7.5
pyramid_net	1,233	43,142	52.6M	0	7	0	100.0%	37.9	7	0	100.0%	38.7	7	0	100.0%	39.2
sbn	1,108	11,262	2.21M	0	42	3	93.3%	4.1	42	3	93.3%	4.1	26	19	57.8%	3.7
sbnrebar	1,108	11,262	2.21M	0	187	2	98.9%	9.8	187	2	98.9%	9.8	107	82	56.6%	8.8
sbndynamicrebar	1,108	31,530	2.61M	0	194	2	99.0%	18.7	194	2	99.0%	19.2	114	82	58.2%	17.9
sbngumbel	1,108	2,070	1.98M	0	78	2	97.5%	4.6	78	2	97.5%	4.6	46	34	57.5%	4.0
audioset	405	699	216M	0	2	0	100.0%	2.9	2	0	100.0%	2.9	1	1	50.0%	2.9
learning_to_remember	702	1,027	4.30M	0	6	0	100.0%	3.1	6	0	100.0%	3.1	6	0	100.0%	3.1
neural_gpu1	2,401	5,080	2.68M	0	53	0	100.0%	5.5	53	0	100.0%	5.5	53	0	100.0%	5.6
neural_gpu2	2,401	2,327	1.35M	0	38	0	100.0%	4.2	38	0	100.0%	4.2	38	0	100.0%	4.2
ptn	1,713	23,636	145M	0	351	0	100.0%	14.8	351	0	100.0%	15.0	351	0	100.0%	14.8
namignizer	262	2,310	652K	0	3	0	100.0%	3.5	3	0	100.0%	3.5	3	0	100.0%	3.5
feelvos	2,955	83,558	83.0M	0	4	0	100.0%	135.4	4	0	100.0%	137.7	4	0	100.0%	132.6
fivo_srnn	5,661	3,514	357K	0	7	11	38.9%	4.2	7	11	38.9%	4.3	7	11	38.9%	4.3
fivo_vrnn	5,661	3,820	365M	0	7	11	38.9%	4.4	7	11	38.9%	4.5	7	11	38.9%	4.5
fivo_ghmm	5,661	2,759	60	0	9	23	28.1%	4.0	9	23	28.1%	4.1	9	23	28.1%	4.1
dcb_var_bnn	2,143	474	36.0K	0	22	0	100.0%	3.0	22	0	100.0%	3.0	22	0	100.0%	3.0
dcb_neural_ban	2,143	186	18.0K	0	4	0	100.0%	2.7	4	0	100.0%	2.7	4	0	100.0%	2.7
dcb_bb_alpha_nn	2,143	11,180	36.0K	0	163	2	98.8%	8.2	163	2	98.8%	8.2	163	2	98.8%	8.4
dcb_rms_bnn	2,143	186	18.0K	0	4	0	100.0%	2.7	4	0	100.0%	2.7	4	0	100.0%	2.7
adversarial_crypto	133	676	8.14K	0	6	0	100.0%	3.0	6	0	100.0%	3.0	6	0	100.0%	3.0
sentiment_analysis	130	334	4.39M	0	3	1	75.0%	2.7	3	1	75.0%	2.7	3	1	75.0%	2.7
next_frame_prediction	493	2,820	6.70M	1	6	0	100.0%	4.0	6	0	100.0%	4.1	6	0	100.0%	4.0
minigo	3,774	929	34.4K	1	0	0	100.0%	3.0	0	0	100.0%	3.0	0	0	100.0%	3.0
compression_entropy_coder	2,000	15,709	20.0K	0	13	0	100.0%	9.7	13	0	100.0%	10.0	13	0	100.0%	9.8
lfads	2,898	51,853	928K	202	213	3	99.3%	48.7	213	3	99.3%	49.5	213	3	99.3%	48.6
lm_1b	3,81	2,926	1.04G	0	1	0	100.0%	4.0	1	0	100.0%	4.0	1	0	100.0%	4.0
swivel	1,449	279	36.0K	0	1	0	100.0%	2.7	1	0	100.0%	2.7	1	0	100.0%	2.7
skip_thought	1,129	6,800	377M	0	15	0	100.0%	5.7	15	0	100.0%	5.8	15	0	100.0%	5.7
video_prediction	462	48,148	41.6M	32	288	0	100.0%	30.7	288	0	100.0%	30.6	288	0	100.0%	30.5
gan_mnist	806	2,664	39.7M	0	3	0	100.0%	3.7	3	0	100.0%	3.8	3	0	100.0%	3.7
gan_cifar	510	3,784	43.3M	0	17	0	100.0%	4.5	17	0	100.0%	4.5	17	0	100.0%	4.5
gan_image_compression	444	4,230	35.5M	0	17	0	100.0%	4.7	17	0	100.0%	4.7	17	0	100.0%	4.7
vid2depth	2,502	35,072	99.6M	0	132	48	73.3%	21.2	132	48	73.3%	21.9	132	48	73.3%	21.5
domain_adaptation	3,079	6,010	7.01M	0	28	0	100.0%	5.6	28	0	100.0%	5.6	25	3	89.3%	5.7
delf	3,683	2,712	9.10M	0	10	0	100.0%	5.1	10	0	100.0%	5.1	10	0	100.0%	5.0
Total	—	—	—	313	2760	230	93.0%	691.1	2564	426	87.1%	694.9	2349	641	80.6%	688.7

Since our approach does not have false negatives by design, we omit this column in reporting our evaluation results.

4.5 Implementation and Hardware Platform

We have implemented our DEBAR tool in Python. All of our measurements are performed on a server running Ubuntu 16.04.6 LTS with a GeForce GTX 1080 Ti GPU and i7-8700K CPU running at 3.70GHz.

4.6 RQ1: Effectiveness of DEBAR

4.6.1 Setup. To answer RQ1, we invoke DEBAR on the two datasets and check the number of true/false positives/negatives, accuracy, and execution time (in seconds).

Table 2: Operations to Check

Operations	Unsafe Constraints
Exp(x), Expm1(x)	$\log(Mf) < \overline{x}$
Log(x)	$\underline{x} < mf$
Log1p(x)	$\underline{x} + 1 < mf$
RealDiv(y, x)	$\underline{x} < mf \wedge \overline{x} > -mf$
Reciprocal(x)	$\underline{x} < mf \wedge \overline{x} > -mf$
Sqrt(x)	$\underline{x} \leq -mf$
Rsqrt(x)	$\underline{x} < mf$

4.6.2 Results. Columns 5–9 of Table 1 show the results. We make the following observations about DEBAR.

- It detects all known numerical errors on the 9 architectures in the first dataset, with zero false positive.
- It detects 299 previously unknown operations that may lead to numerical errors in the real-world architectures from the second dataset. Note that a numerical bug can trigger multiple numerical errors at different operations, e.g., failing to normalize an input tensor that is used in multiple operations.
- It correctly classifies 3,073 operations with only 230 false positives, achieving accuracy of 93.0%.
- It is scalable to handle the real-world architectures, all of which are analyzed in 3 minutes, and the average time is 12.1 seconds.

To understand why DEBAR generates some false positives, we investigate the false positives (FPs) and find the following reasons.

- Some operations depend on an argument to index the tensor elements for the operations. For example, function `gather` returns elements in a tensor based on an argument that specifies the elements' indexes. Since we do not know beforehand which indexes are subject to an operation, we merge the intervals at all possible indexes, leading to imprecision. 116 FPs belong to this category.
- Our affine relation analysis works on only linear expressions. When a non-linear operation is used, we create a new abstract element, leading to imprecision. 15 FPs belong to this category.
- 48 FPs belong to both of the preceding two categories.
- For while loops in RNNs, we do not use tensor partitioning and elementwise affine equality relations but use the classic Kleene iteration together with the widening operator [4] in the interval abstract domain, leading to imprecision. 50 FPs belong to this category.
- The TensorFlow API used to extract computation graphs fails to analyze the shapes of some tensors, leading to 1 FP.

4.7 RQ2: Study on Tensor Abstraction

4.7.1 Setup. To study three tensor abstraction techniques, we compare array smashing and array expansion with tensor partitioning by fixing the numerical abstraction as affine relation analysis (used together with interval abstraction).

4.7.2 Results. Columns 10–13 of Table 1 show the results of array smashing, and Table 3 shows the results of array expansion. Since array expansion times out on 33 of the subjects with a time budget of 30 minutes, we report only the results on the remaining 24 subjects. From the tables, we make the following observations.

Table 3: Results of Array Expansion

Name	Array Expansion			
	TN	FP	Acc	Time
TensorFuzz	0	0	100%	29.6
GitHub-IPS-6	0	0	100%	3.2
GitHub-IPS-9	0	0	100%	3.1
StackOverFlow-7	0	0	100%	72.4
StackOverFlow-14	0	0	100%	2.4
autoencoder_mnae	1	0	100.0%	105.4
autoencoder_vae	1	0	100.0%	232.6
sbn	42	3	93.3%	397.8
sbnrebar	187	2	98.9%	725.1
sbngumbel	78	2	97.5%	401.2
learning_to_remember	6	0	100.0%	913.5
neural_gpu1	53	0	100.0%	702.8
neural_gpu2	38	0	100.0%	336.8
namignizer	3	0	100.0%	629.9
fivo_srnn	7	11	38.9%	44.1
fivo_ghmm	9	23	28.1%	4.1
dcb_var_bnn	22	0	100.0%	12.3
dcb_neural_ban	4	0	100.0%	4.0
dcb_bb_alpha_nn	163	2	98.8%	155.5
dcb_rms_bnn	4	0	100.0%	4.0
adversarial_crypto	6	0	100.0%	3.6
sentiment_analysis	3	1	75.0%	693.8
mingo	0	0	100.0%	8.0
compression_entropy_coder	13	0	100.0%	340.7

- Compared to array smashing, DEBAR even runs faster, indicating that the overhead of tensor partitioning is so negligible that the overhead is dominated by the random error of execution time, and DEBAR successfully eliminates 196 more false positives, improving the (total) accuracy from 87.1% to 93.0%.
- Compared to array expansion, the analysis of DEBAR runs seconds to hundreds of seconds faster, and does not lose any accuracy on all the 24 subjects that array expansion can analyze within the time budget of 30 minutes.

These observations confirm that tensor partitioning is more effective than the other two tensor abstraction techniques.

4.8 RQ3: Study on Numerical Abstraction

4.8.1 Setup. To study two numerical abstraction techniques, we compare sole interval abstraction and affine relation analysis with interval abstraction by fixing the tensor abstraction as tensor partitioning.

4.8.2 Results. Columns 14–17 of Table 1 show the results of DEBAR and sole interval abstraction. DEBAR has a negligible overhead (0.3% on average) and eliminates 411 false positives, improving the

accuracy from 80.6% to 93.0% in total. These observations indicate that the affine relation analysis is effective and substantially contributes to the overall effectiveness.

4.9 Threats to Validity

The threat to internal validity mainly lies in the implementation of our approach—whether our implementation correctly captures our approach. To alleviate the threat, we have manually checked all the warnings reported by our approach, and analyzed the reasons for the false positives, validating the implementation to some extent.

The threat to external validity lies in the representativeness of the subjects. In particular, the proportion of false warnings among all warnings heavily depends on the number of numerical bugs in the subjects and may not be generalizable. On the other hand, the accuracy is more generalizable and thus we choose accuracy as part of the metrics in our evaluation.

The threat to construct validity is mainly that we have defined the input range and parameter range, and real users may set different ranges from us. To alleviate this threat, we take a very conservative approach such that real users are likely to set only smaller ranges rather than larger. To further understand the effect of these ranges, we conduct two additional experiments to understand the effect of removing these ranges. We find that, after removing all the input ranges, the accuracy drops 6.2 percentage points, and after removing all the parameter ranges, the accuracy drops 6.4 percentage points. The results suggest that the ranges do affect the accuracy of the DEBAR tool. However, the effect is relatively small, and our conclusion still holds in general, even if different ranges are specified.

5 RELATED WORK

Static Analysis for TensorFlow Programs. Ariadne [7] can detect errors at code creation time for TensorFlow programs in Python by applying a static analysis framework (WALA). Unlike DEBAR, Ariadne cannot detect bugs at the architecture level. Moreover, Ariadne targets to infer the shapes of tensors and builds a type system for analyzing tensor shapes. Since Ariadne does not track the tensor values, it cannot be applied to detect numerical bugs.

Static Analysis of DL Models. Multiple approaches have been proposed to statically analyze DL models. Reluplex [17] uses Satisfiability Modulo Theory (SMT) to verify properties of DNNs. Dutta et al. [9] proposed an output range analysis for DNNs using a local gradient search and mixed-integer linear programming (MILP). Lomuscio et al. [19] used linear programming to analyze the reachability of DNNs. A recent study [10] shows that these approaches cannot scale up to large DL models due to the scalability issue of existing constraint solvers.

Other work built static analyzers based on abstract interpretation. Gehr et al. [10] proposed AI^2, which deploys abstract interpretation (zonotope domain) to prove safety properties of DNNs. Singh et al. [27] proposed DeepPoly for using floating-point polyhedra and affine transformations to improve the scalability and precision of analysis. Singh et al. [28] proposed RefineZero for using the zonotope domain and MILP to further improve the precision of analysis. Li et al. [18] used the symbolic propagation technique to improve precision. Unlike DEBAR, these approaches aim to analyze neural network models precisely, so these approaches all adapt the array expansion strategy, where each element is instantiated as a scalar variable. As shown by our evaluation, this strategy is not efficient enough to identify numerical bugs before training. On the other hand, DEBAR incorporates a novel tensor abstraction, maps tensor partitions to abstract elements, and discovers linear equality relations between partitions.

Testing DL Models. Quite some previous work on testing DL models focuses on test coverage criteria for DL models [20, 21, 25, 29–31]. For example, based on coverage criteria, Odena et al. [24] proposed TensorFuzz for using coverage-guided fuzzing to test DL models. Such previous work focuses on DL models and does not detect numerical bugs before training.

Adversarial examples are often viewed as revealing vulnerabilities of DL models, and many approaches focus on finding adversarial examples. Popular adversarial attack approaches such as FGSM, C&W, and PGD [2, 11, 22] use gradient-based techniques to generate adversarial examples guided by objective functions whose inputs are the parameters. These approaches cannot be easily adapted to test DL architectures as the objective functions cannot be computed without parameters.

Testing DL Libraries. CRADLE [26] was proposed to detect and locate bugs in deep learning libraries. In contrast, our DEBAR approach targets at DL architectures instead of DL libraries.

Array Analysis in Imperative Programs. Our approach is inspired by the existing approaches of abstract interpretation for analyzing arrays, in particular, array partitioning [6, 12, 13]. Compared with the existing approach of array partitioning, we are the first to generalize this approach from arrays to tensors, employ an affine relation analysis for capturing affine equality relations among partitions, and design abstract tensor operations for DL architectures such as the abstract operation for ReLU.

6 CONCLUSION

We have proposed a static analysis approach to detect numerical bugs in neural architectures. We specially designed tensor partitioning and affine relation analysis (used together with interval abstraction) over partitions for our approach, and implemented them as DEBAR. We evaluated our approach on two datasets with various settings on tensor abstraction and numerical abstraction techniques, and the results show that (1) DEBAR is effective to detect numerical bugs in real-world neural architectures; and (2) two specially designed abstraction techniques are essential for improving the scalability and accuracy of detecting numerical bugs.

ACKNOWLEDGMENTS

This work is supported by the National Key Research and Development Program of China under Grant No. SQ2019YFE010068, the National Natural Science Foundation of China under Grant Nos. 61922003, 61932021, 61872445, and MSRA Collaborative Research Grant.

REFERENCES

[1] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. 2003. A Static Analyzer for Large Safety-critical Software. In *Proceedings of the 2003 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2003, San Diego, California, USA, June 9-11, 2003*. 196–207. https://doi.org/10.1145/781131.781153

[2] Nicholas Carlini and David A. Wagner. 2017. Towards Evaluating the Robustness of Neural Networks. In *Proceedings of the 2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*. 39–57. https://doi.org/10.1109/SP.2017.49

[3] Patrick Cousot and Radhia Cousot. 1976. Static Determination of Dynamic Properties of Programs. In *Proceedings of the 2nd International Symposium on Programming*. Dunod, Paris, France, 106–130.

[4] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conference Record of the 4th ACM Symposium on Principles of Programming Languages, POPL 1977, Los Angeles, California, USA, January 1977*. 238–252. https://doi.org/10.1145/512950.512973

[5] Patrick Cousot and Radhia Cousot. 1977. Static Determination of Dynamic Properties of Generalized Type Unions. In *Proceedings of an ACM Conference on Language Design for Reliable Software, Raleigh, North Carolina, USA*. ACM, 77–94. https://doi.org/10.1145/800022.808314

[6] Patrick Cousot, Radhia Cousot, and Francesco Logozzo. 2011. A Parametric Segmentation Functor for Fully Automatic and Scalable Array Content Analysis. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*. ACM, 105–118.

[7] Julian Dolby, Avraham Shinnar, Allison Allain, and Jenna Reinen. 2018. Ariadne: Analysis for Machine Learning Program. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages, MAPL@PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*. ACM, 1–10. https://doi.org/10.1145/3211346.3211349

[8] Souradeep Dutta, Xin Chen, and Sriram Sankaranarayanan. 2019. Reachability Analysis for Neural Feedback Systems using Regressive Polynomial Rule Inference. In *Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control, HSCC 2019, Montreal, QC, Canada, April 16-18, 2019*. ACM, 157–168. https://doi.org/10.1145/3302504.3311807

[9] Souradeep Dutta, Susmit Jha, Sriram Sankaranarayanan, and Ashish Tiwari. 2018. Output Range Analysis for Deep Feedforward Neural Networks. In *Proceedings of the 10th NASA Formal Methods Symposium, NFM 2018, Newport News, VA, USA, April 17-19, 2018*. 121–138. https://doi.org/10.1007/978-3-319-77935-5_9

[10] Timon Gehr, Matthew Mirman, Dana Drachsler-Cohen, Petar Tsankov, Swarat Chaudhuri, and Martin T. Vechev. 2018. AI2: Safety and Robustness Certification of Neural Networks with Abstract Interpretation. In *Proceedings of the 2018 IEEE Symposium on Security and Privacy, SP 2018, San Francisco, California, USA, May 21-23, 2018*. 3–18. https://doi.org/10.1109/SP.2018.00058

[11] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. 2015. Explaining and Harnessing Adversarial Examples. In *Proceedings of the 3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015*. http://arxiv.org/abs/1412.6572

[12] Denis Gopan, Thomas W. Reps, and Shmuel Sagiv. 2005. A Framework for Numeric Analysis of Array Operations. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*. ACM, 338–350.

[13] Nicolas Halbwachs and Mathias Péron. 2008. Discovering Properties about Arrays in Simple Programs. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2008, Tucson, AZ, USA, June 7-13, 2008*. 339–348. https://doi.org/10.1145/1375581.1375623

[14] Xiaowei Huang, Marta Kwiatkowska, Sen Wang, and Min Wu. 2017. Safety Verification of Deep Neural Networks. In *Proceedings of the 29th International Conference on Computer-Aided Verification, CAV 2017, Heidelberg, Germany, July 24-28, 2017*. 3–29. https://doi.org/10.1007/978-3-319-63387-9_1

[15] Md Johirul Islam, Giang Nguyen, Rangeet Pan, and Hridesh Rajan. 2019. A Comprehensive Study on Deep Learning Bug Characteristics. In *Proceedings of the 2019 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*. 510–520. https://doi.org/10.1145/3338906.3338955

[16] Michael Karr. 1976. Affine Relationships among Variables of a Program. *Acta Inf.* 6, 2 (June 1976), 133–151. https://doi.org/10.1007/BF00268497

[17] Guy Katz, Clark W. Barrett, David L. Dill, Kyle Julian, and Mykel J. Kochenderfer. 2017. Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks. In *Proceedings of the 29th International Conference on Computer-Aided Verification, CAV 2017, Heidelberg, Germany, July 24-28, 2017*. 97–117. https://doi.org/10.1007/978-3-319-63387-9_5

[18] Jianlin Li, Jiangchao Liu, Pengfei Yang, Liqian Chen, Xiaowei Huang, and Lijun Zhang. 2019. Analyzing Deep Neural Networks with Symbolic Propagation: Towards Higher Precision and Faster Verification. In *Proceedings of the 26th Static Analysis Symposium, SAS 2019, Porto, Portugal, October 8-11, 2019 (Lecture Notes in Computer Science, Vol. 11822)*. Springer, 296–319. https://doi.org/10.1007/978-3-030-32304-2_15

[19] Alessio Lomuscio and Lalit Maganti. 2017. An Approach to Reachability Analysis for Feed-forward ReLU Neural Networks. *CoRR* abs/1706.07351 (2017). arXiv:1706.07351 http://arxiv.org/abs/1706.07351

[20] Lei Ma, Felix Juefei-Xu, Minhui Xue, Bo Li, Li Li, Yang Liu, and Jianjun Zhao. 2019. DeepCT: Tomographic Combinatorial Testing for Deep Learning Systems. In *Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2019, Hangzhou, China, February 24-27, 2019*. 614–618. https://doi.org/10.1109/SANER.2019.8668044

[21] Lei Ma, Felix Juefei-Xu, Fuyuan Zhang, Jiyuan Sun, Minhui Xue, Bo Li, Chunyang Chen, Ting Su, Li Li, Yang Liu, Jianjun Zhao, and Yadong Wang. 2018. DeepGauge: Multi-granularity Testing Criteria for Deep Learning Systems. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*. 120–131. https://doi.org/10.1145/3238147.3238202

[22] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. 2018. Towards Deep Learning Models Resistant to Adversarial Attacks. In *Proceedings of the 6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018*. https://openreview.net/forum?id=rJzIBfZAb

[23] Nina Narodytska, Shiva Prasad Kasiviswanathan, Leonid Ryzhyk, Mooly Sagiv, and Toby Walsh. 2018. Verifying Properties of Binarized Deep Neural Networks. In *Proceedings of the 32rd AAAI Conference on Artificial Intelligence, AAAI 2018, New Orleans, Louisiana, USA, February 2-7, 2018*. 6615–6624. https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/16898

[24] Augustus Odena, Catherine Olsson, David Andersen, and Ian J. Goodfellow. 2019. TensorFuzz: Debugging Neural Networks with Coverage-Guided Fuzzing. In *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, Long Beach, California, USA, 9-15 June 2019*. 4901–4911. http://proceedings.mlr.press/v97/odena19a.html

[25] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. 2017. DeepXplore: Automated Whitebox Testing of Deep Learning Systems. In *Proceedings of the 26th Symposium on Operating Systems Principles, SOSP 2017, Shanghai, China, October 28-31, 2017*. ACM, 1–18. https://doi.org/10.1145/3132747.3132785

[26] Hung Viet Pham, Thibaud Lutellier, Weizhen Qi, and Lin Tan. 2019. CRADLE: Cross-Backend Validation to Detect and Localize Bugs in Deep Learning Libraries. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*. IEEE / ACM, 1027–1038. https://doi.org/10.1109/ICSE.2019.00107

[27] Gagandeep Singh, Timon Gehr, Markus Püschel, and Martin Vechev. 2019. An Abstract Domain for Certifying Neural Networks. *Proc. ACM Program. Lang.* 3, POPL, Article 41 (Jan. 2019), 30 pages. https://doi.org/10.1145/3290354

[28] Gagandeep Singh, Timon Gehr, Markus Püschel, and Martin T. Vechev. 2019. Boosting Robustness Certification of Neural Networks. In *Proceedings of the 7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. https://openreview.net/forum?id=HJgeEh09KQ

[29] Youcheng Sun, Xiaowei Huang, and Daniel Kroening. 2018. Testing Deep Neural Networks. *CoRR* abs/1803.04792 (2018). arXiv:1803.04792 http://arxiv.org/abs/1803.04792

[30] Youcheng Sun, Min Wu, Wenjie Ruan, Xiaowei Huang, Marta Kwiatkowska, and Daniel Kroening. 2018. Concolic Testing for Deep Neural Networks. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*. 109–119. https://doi.org/10.1145/3238147.3238172

[31] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. 2018. DeepTest: Automated Testing of Deep-neural-network-driven Autonomous Cars. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*. 303–314. https://doi.org/10.1145/3180155.3180220

[32] Shiqi Wang, Kexin Pei, Justin Whitehouse, Junfeng Yang, and Suman Jana. 2018. Formal Security Analysis of Neural Networks using Symbolic Intervals. In *Proceedings of the 27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*. USENIX Association, 1599–1614. https://www.usenix.org/conference/usenixsecurity18/presentation/wang-shiqi

[33] Jie M. Zhang, Mark Harman, Lei Ma, and Yang Liu. 2019. Machine Learning Testing: Survey, Landscapes and Horizons. *CoRR* abs/1906.10742 (2019). arXiv:1906.10742 http://arxiv.org/abs/1906.10742

[34] Yuhao Zhang, Yifan Chen, Shing-Chi Cheung, Yingfei Xiong, and Lu Zhang. 2018. An Empirical Study on TensorFlow Program Bugs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*. 129–140. https://doi.org/10.1145/3213846.3213866