Detecting Numerical Bugs in Neural Network Architectures

Anonymous Author(s)

ABSTRACT

Detecting bugs in deep learning software at the architecture level provides additional benefits that detecting bugs at the model level does not. 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 (element-wise) 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. Experiment 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.

KEYWORDS

Neural Network, Static Analysis, Numerical Bugs

ACM Reference Format:

Anonymous Author(s). 2019. 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 2020).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

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 perform 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.

In order to avoid unexpected or incorrect behaviors in DL software systems, it is necessary to detect bugs in their neural architectures. Although various approaches [7–9, 13, 15–18, 20–22, 24–29] have been proposed to test or verify DL models, they do not address the needs of two types of stakeholders: (1) architecture vendors who design and publish neural architectures to be used by other users, and (2) developers who use neural architectures to train and deploy a model based on their own training dataset.

- Architecture vendors need to provide quality assurance for their neural architecture. It is inadequate for vendors to verify the architecture with specific instantiated models which are datasets-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 [32]. 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 happen frequently during software system development and deployment, e.g., when the new feedback data is collected from users [31].

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 behavior when certain inputs are encountered during real system usage [21].

Detecting numerical bugs via testing is either too challenging at the architecture level or too late at the model level as revealed in earlier empirical studies [14, 21, 32]. 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.

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

ESEC/FSE 2020, 8 - 13 November, 2020, Sacramento, California, United States

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

To detect numerical bugs at the architecture level, in this paper, we propose to use static analysis under the framework of abstract interpretation, 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, 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 them, array expansion, array smashing and interval abstraction are adapted from existing abstraction techniques for imperative programs [1, 4]. 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 the accuracy of our approach on two datasets: a set of 9 architectures with known numerical bugs collected by existing studies [21, 32], and a set of 48 large real-world neural architectures. 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. In terms of scalibility, experiment 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%, (sole) interval abstraction 80.6% which demonstrates 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 real-world code snippet that creates rectangles and calculates the reciprocals of their areas[2].

```
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 a rectangle.
6  bottomLeft = center - offset
7  topRight = center + offset
8  rectangle = tf.concat([bottomLeft, topRight], axis=1)
9
10 # Calculate the reciprocal of its area.
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 the bottom left points and the top right points are calculated from the central points and the offsets (Lines 6-7). Based on them, rectangles are created by concatenating the two points into a 4*100 tensor (Line 8). The second part calculates the reciprocals of the areas. First, the bottom, left, top, right coordinates are extracted from `rectangle` (Line 11), which are all 1*100-shape tensors. 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 is an effective solution to statically consider all possible values of variables [3]. 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, there are mainly two aspects that we need to consider. First, the numeric values and arithmetic computations need to be abstracted. Second, the tensors need to be abstracted. We first discuss three abstract techniques adapted from existing work for analyzing imperative programs, and then describe two new techniques we proposed for analyzing neural architectures.

2.1 Interval abstraction

Interval abstraction [4] is a popular abstract interpretation technique for abstracting numeric values, where each scalar variable v is represented by an interval, indicating the lower bound and

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

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.

`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, for example, by restricting the argument from assuming 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. Assume 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 is a basic technique for abstracting arrays 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 actually performed. Mapping it 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 this example we need to record 200 intervals for `center` and 200 intervals for `offset`, significantly 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 is an alternative abstraction technique which uses one abstract element to represent all elements in a tensor. In this way, the number of abstract elements is greatly reduced. Mapping it for tensor abstraction with interval abstraction, we use one range to cover the 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, we can get the interval of `area` is $[-36, 36]$, which is more imprecise than the previous approach. 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 numeric 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 indices for each partition. We use $\mathbb{I}_A$ to denote the index ranges of the partition A, and use $*$ to denote all indexes in a dimension are included. 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. And we use $\sigma(a)$ to denote its corresponding interval. With these summary variables, we maintain affine equality relations among these summary variables.

In this 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.

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

The calculation is dissimilar to interval abstraction with array smashing in two ways for the example. The first difference in calculation occurs on Line 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 them from the corresponding partitions in `rectangle`.

The second difference in calculation occurs at Line 12. 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 can not get any affine equality relations for `area`. Hence, we introduce a summary variable a for the whole `area`, and compute its interval range. Then, we get an interval $[0, 16]$ for `area`, which is actually the ground-truth interval range of `area`.

3 APPROACH

In this section, we first describe our static analysis approach comprising the tensor abstraction and the numerical abstraction. We then show how to abstract tensor operations under two abstraction techniques, tensor partitioning and affine relation analysis, designed for neural architectures. We also discuss the initial intervals for input ranges and parameter ranges.

3.1 Abstractions for Neural Architectures

Tensors in a neural architecture are represented by multi-dimensional numerical arrays in DL frameworks. To ease presentation, we illustrate our approach using vectors (one-dimensional tensors) and matrices (two-dimensional tensors). Our approach is generalizable to multi-dimensional tensors.

3.1.1 Tensor Abstraction. We mainly leverage existing array abstractions designed for regular programs (e.g., C programs) and adapt them to fit for our new context of neural architectures. There are two basic array abstractions widely used in abstract interpretation based program analysis: array smashing and array expansion [1]. Using array smashing, all the elements of an array A are subsumed by the same scalar variable a, while each element of A is instantiated as a separate scalar variable using array expansion. Array expansion cannot scale up for large DNNs since we may have to instantiate millions of elements in tensors. Array smashing is quite lightweight and suitable for analyzing large tensors in DNNs. Hence, we use array smashing as our basic abstract domain to deal with tensors. However, purely using array smashing, the analysis may be not precise enough.

To this end, considering the features of operations over tensors, we introduce a new granularity of array abstraction, named **tensor partitioning**, which is a form of array partitioning (also named array segmentation) [5, 11, 12] 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 indices 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 indices in $\mathbb{I}_{A_i}$ are indices of the corresponding elements in tensor A, while we sometimes use $\mathbb{I}_{A_i.shape}$ to represent the indices of the corresponding elements in sub-tensor A_i, where $A_i.shape$ means a tuple of integers giving the size of the sub-tensor A_i along each dimension. In our motivating example, R_1, R_2 are two partitions 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 indices 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) [5, 11, 12], since 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 two kinds of information for these partitions, which will be detailed in the next two subsections.

3.1.2 Interval Abstraction. The range information of variable values is the lower and upper bound of the values that a variable may take, which is quite useful in detecting numerical bugs. We use the classic *interval* abstract domain to infer the value range for scaler variables in the DL programs and also for those auxiliary abstract summary variables introduced by array smashing and tensor partitioning. The interval abstract domain is known for its scalability due to its linear time and space complexity, and thus is fit to analyze large DNNs.

3.1.3 Elementwise Affine Relations between Tensor Partitions. Elementwise tensor operations, such as addition and subtraction, are common in DNN implementations. Thus, we also maintain elementwise symbolic relations among tensor partitions. More clearly, we consider the elementwise relations among tensor partitions $A_1, A_2, \ldots, A_m$ which have the same shape but may take from different tensors, in the form of

$$\forall i \in \mathbb{I}_{(A_1.shape)}.\psi(A_1[i], A_2[i], \ldots, A_m[i])$$

where ψ denotes (equality or inequality) relations among its parameters. We could use relational numerical abstract domains (such as polyhedra) to infer (inequality) relations. However, since many tensor operations as well as affine transformations (e.g., assignments) in DNN implementations induce affine equality relations, in this paper, we only consider the elementwise affine equality relations among tensor partitions. In addition, affine equality relations are cheap to infer, and thus is fit to analyze large DNNs.

For the sake of simplicity, since for each partition A_i, we will introduce an abstract summary variable a_i to subsume the elements

in A_i, the above form can be simply represented as

$$\sum_{i>0} \omega_i a_i = \omega_0$$

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

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

$$b_j = a_i^{ReLU}$$

which means

$$\forall k \in \mathbb{I}_{(B_j.shape)}.B_j[k] = A_i^{ReLU}[k]$$

3.2 Abstracting Tensor Operations

Now, we 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 the supplementary material.

3.2.1 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 be not partitioned in the same way, we should first align the partitions of A and those of B, such that $\forall i \in [0, \mathcal{N}_A].\mathbb{I}_{A_i} = \mathbb{I}_{B_i}$ 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 are aligned with A, B.

After that, for each partition C_i of the resulting tensor C, 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

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

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 input one-dimensional tensors A and B are partitioned into 2 partitions, and

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

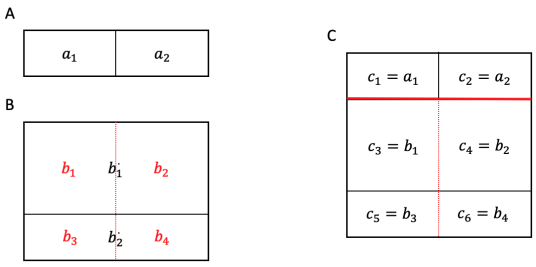

Figure 1: Concatenating tensors (horizontally)

After aligning partitions of A and B, we have

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

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

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

3.2.2 Concatenate. In DNN implementations, tensors can be concatenated along certain dimension p. An assignment statement $C = Concatenate(A, B, p)$ means 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 input 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 do this, 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 $\mathcal{N}_A, \mathcal{N}_B$ respectively denote the number of partitions of A, B after aligning.

For simplicity of presentation, here we assume 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

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

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

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

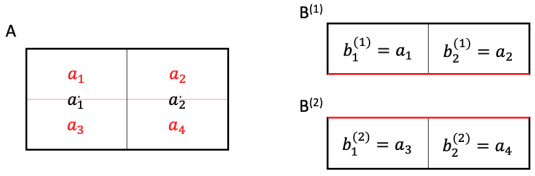

Figure 2: Splitting tensors (vertically)

which means

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

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:

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

3.2.3 Split. A tensor can be split into sub-tensors along a certain dimension. More clearly, a statement $(B^{(1)}, \ldots, B^{(n)}) = split(A, n, p)$ means A is split along dimension p into n smaller tensors (stored in $B^{(1)}, \ldots, B^{(n)}$), wherein it 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 aligning. 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 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 2 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 $i' = (j-1) * 2 + i$.

3.3 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 real world, the input may fall into only a small range. For example, a RGB value of an image falls into $[0, 255]$. Similarly, the parameters of a neural network may also fall into a small range. For example, many neural architecture 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

4.1 Research Questions

Our evaluation aims to answer the following research questions.

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

The first research question concerns about the performance of DEBAR. The second question studies three tensor abstraction techniques. The third question studies two numerical abstraction techniques.

4.2 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: eight architectures were collected by a previous empirical study on TensorFlow bugs [32] and one architecture was obtained from the study that proposes and evaluates TensorFuzz [21].

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 TensorFlow Models repository[3]. The whole collections contains 66 projects implemented in TensorFlow by researchers and developers for different tasks in various domains including computer vision, natural language processing, speech recognition, adversarial machine learning. We first filter out the projects which are not related to specific neural architectures like API framework or optimizer. For the rest of them, we further filter out those we fail to reproduce and generate its computation graph due to incomplete documentation or complicate configuration. 32 projects remain after filtering, and each of them contains one or more neural architectures. Overall, our second dataset contains a great diversity of neural architectures like CNN(Convolutional Neural Network), RNN(Recurrent Neural Network), GAN(Generative Adversarial Network), HMM(Hidden Markov Model) and so on. Please note that we have no knowledge about whether the architectures in this dataset contain numerical bugs when collecting the dataset.

For every architecture in two datasets, we extract the computation graph by using a TensorFlow API. Each extracted computation graph is represented by a Protocol Buffer file[4], which provides the operations (nodes) and the data flow relations (edges). We provide all 48 computation graphs in our supplementary material.

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

4.3 Setups of Input Range and Parameter Range

In our experiments, we conservatively provide the input ranges. Following Sect. 3.3, we get the initial input ranges from the physical meaning of inputs, and derive input range information form the preprocessing programs. If we fail to provide input ranges by above two steps, we set the input ranges to `[FLOAT_MIN, FLOAT_MAX]`.

We determine the parameter ranges by 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 code.

4.4 Unsafe Operations to Check

By investigating the dataset in the empirical study [32], we collect a list of unsafe operations shown in Table 2. Those operations have the most frequent occurrences and have high potential to cause numerical errors. In this paper, we use our abstract interpretation based static analysis approach, 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 constraints listed in Table 2. If the constraints are satisfiable, our checker will issue an alarm, which means the operation may cause numerical errors. Otherwise, it means that the operation is safe. In Table 2, Mf and mf respectively denote the largest non-infinity floating-point number and the smallest nonzero positive floating-point number that the used floating-point format (e.g., 32-bit, 64-bit) can represent exactly.

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

4.5 Measurements

Our approach checks every operation that may lead to a numerical error and determines if a warning should be reported. To measure the performance of our approach, we treat it as a classifier that classifies whether each operation is buggy, and evaluates its performance 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 reference user patches to determine whether a warning is a bug. For the second dataset,

- 204 true positives are confirmed by executing the architecture using designed inputs and parameters to trigger the numerical errors.
- 52 true positives are confirmed by the developers provided fixes (not merged yet) in the issue discussions.
- 43 true positives are confirmed when two authors do reasoning on the computation graph separately and both conclude that they are true positive.

We report all the true positive warnings in the supplementary material. Otherwise, we regard the warnings as false positives. Since our approach does not have false negatives by design, we omit this column in reporting our experimental results.

[3]https://github.com/tensorflow/models/blob/13e7c85d521d7bb7cba0bf7d743366f7708b9df7/research

[4]https://en.wikipedia.org/wiki/Protocol_Buffers

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

4.6 Implementation and Hardware Platform

We have implemented our tool DEBAR in Python, which is available on Github[5]. All of our measurements were 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.

[5]Use the anonymous link for review. https://anonymous.4open.science/r/1bf6b4c5-18d3-4105-b8ba-f001c915d112

4.7 RQ1: Performance of DEBAR

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

4.7.2 Results. The results are shown in columns 5–9 of Table 1. We make the following observations about DEBAR from the dataset.

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)	$\underline{x} < mf$

- It can detect all known numerical errors on the 9 architectures, with zero false positives.
- It detects 299 previously unknown operations that may lead to numerical errors in the real architectures. 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, giving an accuracy of 93.0%.
- It is scalable to handle real-world architectures, where all architectures are analyzed in 3 minutes, and the average time is 12.1 seconds.

To understand why DEBAR generates some false positives, we investigated all false positives, and found 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 a parameter that specifies their 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 case.
- Our affine relation analysis only works on linear expressions. When a non-linear operation is used, we create a new abstract element, leading to imprecision. 15 FPs belong to this case.
- 48 FPs belong to both of the above two cases.
- For while loops in RNNs, we do not maintain any elementwise affine equality relations and use the classic Kleene iteration together with the widening operator [3] in the interval abstract domain, leading to imprecision. 50 FPs belong to this case.
- The TensorFlow API used to extract computation graphs fails to analyze the shapes of some tensors, leading to one false positive.

4.8 RQ2: Study on Tensor Abstraction

4.8.1 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).

4.8.2 Results. The results of array smashing are shown in columns 10–13 of Table 1, and the results of array expansion are shown in Table 3. Since array expansion timed out on 33 of the subjects

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

with a time budget of 30 minutes, we only report the results on the remaining 24 subjects. From the tables, we make the following observations.

- Compared to array smashing, DEBAR even runs faster, which indicates that the overhead of tensor partitioning is so negligible that it 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 30 minutes time budget.

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

4.9 RQ3: Study on Numerical Abstraction

4.9.1 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.

4.9.2 Results. The results of DEBAR and sole interval abstraction are shown in columns 14–17 of Table 1. It can be seen that DEBAR has a negligible overhead (0.3% on average), and eliminates 411 false positive cases improving the accuracy from 80.6% to 93.0% in

total. These observations indicate that the affine relation analysis is effective and significantly contributes to the overall performance.

4.10 Threats to Validity

The internal threat to validity mainly lies in the implementation of our approach—whether our implementation correctly captured our approach. On the other hand, we have manually checked all the warnings our approach reported, and analyzed the reason for the false positives, which validated the implementation to some extent.

The external threat to 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 the previous metrics in our experiment.

A construct threat to validity is that we have defined the input range and parameter range, and real users may set different ranges from us. To minimize this threat, we take a very conservative approach, such that real users are likely to only set smaller ranges rather than larger. To further understand the effect of these ranges, we performed two additional experiments to understand the effect of removing these ranges. We found that, after removing all the input ranges, the accuracy dropped 6.2 percentage points, and after removing all the parameter ranges, the accuracy dropped 6.4 percentage points. The results suggest that the ranges do affect the accuracy of the 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 [6] can detect errors at code creation time for TensorFlow programs in Python by applying a static 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 it does not track the tensor values, it cannot be applied to detecting numerical bugs.

Static Analysis of DL Models. Multiple approaches have been proposed to statically analyze DL models. Reluplex [15] proposed using Satisfiability Modulo Theory (SMT) to verify the properties of DNNs. Dutta et al. [8] proposed an output range analysis for DNNs using a local gradient search and mixed integer linear programs (MILP). Lomuscio et al. [16] used linear programming to analyze the reachability of DNNs. It is shown by a recent study [9] that these approaches cannot scale up to large DL models due to the scalability issue of existing constraint solvers.

Other approaches built static analyzers based on abstract interpretation. Gehr et al. [9] proposed AI^2, which deploys abstract interpretation (zonotope domain) to prove safety properties of the DNN. Singh et al. [24] proposed DeepPoly making use of floating point polyhedra and affine transformations to improve the scalability and precision of analysis. In another study, Singh et al. [27] proposed RefineZero using zonotope domain and MILP to further improve the precision of analysis. Yang et al. [30] used the symbolic propagation technique to improve precision. Unlike DEBAR, these approaches aim to perform precise analysis to neural network models, so they all adapt the array expansion strategy, where each element is instantiated as a scalar variable. As shown by our evaluation, such a strategy is not efficient enough to identify numerical bugs before training. On the other hand, DEBAR incorporates the novel tensor abstraction, maps tensor partitions to abstract elements, and discovers linear equality relations between partitions.

Testing DL Models. A number of existing studies focus on testing DL models. In particular, many of them focus on the test coverage criteria for DL models [17, 18, 22, 26–28]. Based on coverage criteria, Odena et al. [21] proposed TensorFuzz that used the coverage-guided fuzzing methods to test DL models. These approaches focus on DL models and do 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, 10, 19] 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 their objective functions cannot be computed with free parameters.

Testing DL Libraries. CRADLE [23] 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 abstraction interpretation for analyzing arrays, in particular, array partitioning [5, 11, 12]. 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 linear equality relations that can be computed directly (in contrast to general inequality relations that require a solver to resolve), and design abstract tensor operations for DL architectures such as the abstract operation for ReLU.

6 CONCLUSION

We 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.

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 ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation 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 *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. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*. 238–252. https://doi.org/10.1145/512950.512973

[4] Patrick Cousot and Radhia Cousot. 1977. Static Determination of Dynamic Properties of Generalized Type Unions. *SIGPLAN Not.* 12, 3 (March 1977), 77–94. https://doi.org/10.1145/390017.808314

[5] 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.

[6] Julian Dolby, Avraham Shinnar, Allison Allain, and Jenna Reinen. 2018. Ariadne: Analysis for Machine Learning Programs. In *Proceedings of the 2Nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages* (Philadelphia, PA, USA) *(MAPL 2018)*. ACM, New York, NY, USA, 1–10. https://doi.org/10.1145/3211346.3211349

[7] 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* (Montreal, Quebec, Canada) *(HSCC '19)*. ACM, New York, NY, USA, 157–168. https://doi.org/10.1145/3302504.3311807

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

[9] 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 *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*. 3–18. https://doi.org/10.1109/SP.2018.00058

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

[11] 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.

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

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

[14] Md Johirul Islam, Giang Nguyen, Rangeet Pan, and Hridesh Rajan. 2019. A comprehensive study on deep learning bug characteristics. In *Proceedings of the 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

[15] 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 *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I*. 97–117. https://doi.org/10.1007/978-3-319-63387-9_5

[16] 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

[17] 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 *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

[18] 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

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

[20] Nina Narodytska, Shiva Prasad Kasiviswanathan, Leonid Ryzhyk, Mooly Sagiv, and Toby Walsh. 2018. Verifying Properties of Binarized Deep Neural Networks. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*. 6615–6624. https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/16898

[21] 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, 9-15 June 2019, Long Beach, California, USA*. 4901–4911. http://proceedings.mlr.press/v97/odena19a.html

[22] 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* (Shanghai, China) *(SOSP '17)*. ACM, New York, NY, USA, 1–18. https://doi.org/10.1145/3132747.3132785

[23] 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*. 1027–1038. https://dl.acm.org/citation.cfm?id=3339633

[24] 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

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

[26] 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

[27] 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

[28] 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

[29] 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 Conference on Security Symposium* (Baltimore, MD, USA) *(SEC'18)*. USENIX Association, Berkeley, CA, USA, 1599–1614. http://dl.acm.org/citation.cfm?id=3277203.3277323

[30] Pengfei Yang, Jiangchao Liu, Jianlin Li, Liqian Chen, and Xiaowei Huang. 2019. Analyzing Deep Neural Networks with Symbolic Propagation: Towards Higher Precision and Faster Verification. *CoRR* abs/1902.09866 (2019). arXiv:1902.09866 http://arxiv.org/abs/1902.09866

[31] 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

[32] 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