Exposing Numerical Bugs in Deep Learning via Gradient Back-Propagation

Ming Yan
College of Intelligence and Computing, Tianjin University
China
yanming@tju.edu.cn

Junjie Chen*
College of Intelligence and Computing, Tianjin University
China
junjiechen@tju.edu.cn

Xiangyu Zhang
Purdue University
USA
xyzhang@cs.purdue.edu

Lin Tan
Purdue University
USA
lintan@purdue.edu

Gan Wang
College of Intelligence and Computing, Tianjin University
China
acmer.wg@gmail.com

Zan Wang
College of Intelligence and Computing, Tianjin University
China
wangzan@tju.edu.cn

ABSTRACT

Numerical computation is dominant in deep learning (DL) programs. Consequently, numerical bugs are one of the most prominent kinds of defects in DL programs. Numerical bugs can lead to exceptional values such as NaN (Not-a-Number) and INF (Infinite), which can be propagated and eventually cause crashes or invalid outputs. They occur when special inputs cause invalid parameter values at internal mathematical operations such as `log()`. In this paper, we propose the first dynamic technique, called GRIST, which automatically generates a small input that can expose numerical bugs in DL programs. GRIST piggy-backs on the built-in gradient computation functionalities of DL infrastructures. Our evaluation on 63 real-world DL programs shows that GRIST detects 78 bugs including 56 unknown bugs. By submitting them to the corresponding issue repositories, eight bugs have been confirmed and three bugs have been fixed. Moreover, GRIST can save 8.79X execution time to expose numerical bugs compared to running original programs with its provided inputs. Compared to the state-of-the-art technique DEBAR (which is a static technique), DEBAR produces 12 false positives and misses 31 true bugs (of which 30 bugs can be found by GRIST), while GRIST only misses one known bug in those programs and no false positive. The results demonstrate the effectiveness of GRIST.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; • **Computing methodologies** → *Machine learning*.

KEYWORDS

Deep Learning Testing, Numerical Bug, Gradient Back-propagation, Search-based Software Testing

ACM Reference Format:
Ming Yan, Junjie Chen, Xiangyu Zhang, Lin Tan, Gan Wang, and Zan Wang. 2021. Exposing Numerical Bugs in Deep Learning via Gradient Back-Propagation. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21), August 23–28, 2021, Athens, Greece.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3468264.3468612

1 INTRODUCTION

In recent years, DL systems have become one of the most popular kinds of software systems and are widely used in various domains, e.g., face recognition [44], autonomous driving [10], and software engineering [11, 12, 14, 52]. A DL system consists of three levels as shown in Figure 1, including the production level (i.e., DL models), program level (i.e., DL programs that are used for building DL models), and infrastructure level (e.g., DL libraries). Bugs in any level could affect the overall quality of the DL system. Therefore, it is necessary to guarantee the quality of DL systems at all the three levels. Currently, a great deal of research has been conducted on the production level by proposing various adversarial input generation methods [9, 21, 30, 38, 49] or designing various testing metrics [29, 34, 39], but there is relatively little attention on the other two levels. Actually, both the program level and the infrastructure level are the basis of the production level since DL models are built based on DL programs by invoking DL libraries, and thus bugs in the former two levels could directly affect the performance of DL models [47, 56]. Therefore, it is critical to guarantee the quality at these two levels. In this paper, we target the program level.

Different from traditional programs, the life-cycle of a DL program consists of not only the traditional coding phase, but also the expensive training phase, in which a large corpus of data is used to train the DL model parameters, and the validation phase, which is analogous to the testing and debugging phase in traditional software development and aims to provide feedback to change training inputs or hyper-parameters to achieve better accuracy. Their erroneous behaviors may have consequences in both the cyber space

[*] Junjie Chen is the corresponding author.

ESEC/FSE '21, August 23–28, 2021, Athens, Greece
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8562-6/21/08…$15.00
https://doi.org/10.1145/3468264.3468612

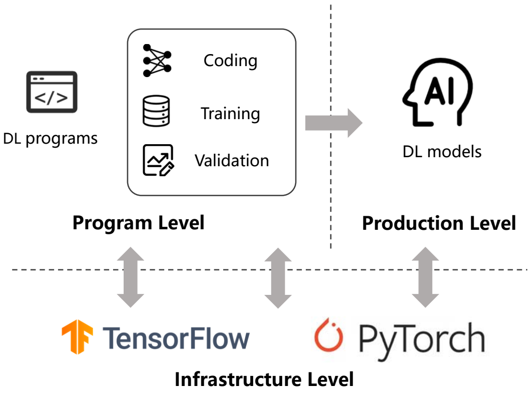


Figure 1: The architecture of DL systems

```python
1 def normalize_frames(m):
2     return [(v - np.mean(v)) / np.std(v) for v in m]
```

(a) TensorFlow program bug example from GitHub[1]

```python
1  def forward(self,x)
2      N, C, H, W = x.size()
3      x = x.transpose(0,1).contiguous().view(C,-1)
4      mu = x.mean(1, keepdim=True)
5      sigma = x.var(1, keepdim=True)
6      # ......
7      x = x - mu
8      x = x / (sigma.sqrt() + self.eps)
9      x = x * self.weight + self.bias
10     x = x.view(C, N, H, W).transpose(0, 1)
11     return x
```

(b) PyTorch program bug example from PyTorch Forums[4]

Figure 2: Examples of numerical bugs in DL programs

and the physical space, some even life-threatening, depending on the application scenarios. Therefore, detecting bugs in DL programs is indeed critical. Following the existing work [56], our work also focuses on numerical bugs in DL programs, since they are one of the most prominent categories of DL program bugs due to the very heavy presence of numerical computation in DL programs. Moreover, numerical bugs could occur at various stages of DL programs, including the data preprocessing stage, training stage, and validation stage.

Numerical bugs in DL programs manifest themselves in the form of "NaN" (meaning that the value is not a number), "INF" (meaning that the value is an infinite number), or crash during the process of training or validation [56]. They are typically caused by mathematical property violations or floating-point representation errors. Once a numerical bug is triggered in computation, it will continue to propagate and eventually lead to invalid outputs. Figure 2a shows an example bug in a TensorFlow program [1], in which NaN appears in the return value of a function `normalize_frames()` at Line 2 when the divisor `np.std(v)` is zero. This bug was not discovered until the program was released. Figure 2b also shows another numerical bug that is not easy to expose in a PyTorch program [4]. Specifically, a PyTorch user reported that she/he encountered NaN when training the DL model, even though she/he had specially added a small value `self.eps` to the denominator at Line 8 to avoid division by zero. However, NaN was still thrown out after running for a period of time. Later, it was found that the program tried to access the derivative of `sigma.sqrt()` when `sigma` was zero. Since `sqrt(x)` has no derivative when `x=0`, an NaN is produced.

Although numerical bugs are prevalent in DL programs, many are very difficult to find, reproduce, and fix. Unlike traditional programs, DL programs require lengthy training (maybe on the scale of days or even months) with a large scale of data in order to achieve good accuracy. The process is dominated by numerical computation. That is, numerical bugs may not be triggered until several hours, days, or even weeks into the training process. These bugs hence may cost developers a high price since the expensive training may have to be redone. Furthermore, these bugs may be non-deterministic, which means that they may or may not manifest themselves during a particular training step. This is because random values are heavily used in DL programs, e.g., in initialization, regularization, and optimization. As such, these bugs may be difficult to reproduce, even though reproduction is the necessary first step for understanding the root cause and fixing it. Therefore, it is very meaningful to expose numerical bugs, confirm them through deterministic reproduction with failure-inducing inputs, and reduce such inputs to minimize debugging efforts.

Recently, Zhang et al. [56] proposed the first static technique, called DEBAR, to detect numerical bugs in TensorFlow programs. Specifically, DEBAR incorporates abstract interpretation to statically analyze whether the value of a variable can violate its valid range in mathematical calculation. Although it has been demonstrated to be effective to some degree, DEBAR suffers from false positives like many static techniques in other domains [6, 17, 24, 46]. Also, like all other static techniques, DEBAR requires manually creating models for third-party libraries that are in other languages or do not have source code. Besides, DEBAR relies on the static computation graph of a DL program, and thus cannot be applicable to DL programs with dynamic computation graphs such as PyTorch programs, which account for a large portion of DL programs in practice. To further guarantee the quality of DL programs, we propose the first *dynamic* technique, called GRIST (**GRa**d**I**ent **S**earch based Numerical Bug **T**riggering), to expose numerical bugs. GRIST gets rid of false positives, does not require modeling third-party libraries, and can be applied to both DL programs with static computation graphs and those with dynamic computation graphs. In particular, GRIST not only points out where a numerical bug is, but also provides a small concrete input that can deterministically trigger the bug within short execution time.

Specifically, we observe that if a numerical bug is not deterministic (meaning that it may or may not be triggered depending on the input and the particular run), it must be directly or transitively related to some external values, which could be training input samples or values generated by random functions (e.g., random initial weights). These external values induce invalid operands at numerical operations (such as division) or invalid parameters to mathematical functions (such as `log()`), causing NaN/INF. While the dataflow from external inputs to the failure points may be highly complex (e.g., through many layers of matrix multiplications, ReLUs, and max-pooling), the underlying infrastructures such as TensorFlow and PyTorch have a powerful mechanism to compute the gradients of arbitrary operands and function parameters regarding external inputs. As such, we do not need to derive the explicit symbolic form of data flow like in [56]. Instead, we leverage the gradients (through back-propagation) to understand how we should change the external values to induce an exception. To realize the idea, we overcome a number of practical challenges. For example, a DL program by default only computes gradients between a loss function and the model weight values (during training) or between a loss function and the input (during adversarial sample generation [35]). In contrast, we need to compute gradients between an arbitrary external value and a parameter of some internal mathematical operation in GRIST. Furthermore, DL program training is different from normal software execution. It takes in a large corpus of inputs through multiple iterations in a random fashion. We need to have a way to supply the mutated inputs (generated by back-propagation) to the training process so that GRIST can induce the failure.

We conducted an experimental study to evaluate the effectiveness of GRIST in exposing numerical bugs and accelerating failure triggering, based on 63 real-world DL programs that are collected from GitHub according to the descending order of GitHub search relevance with operations vulnerable to numerical bugs (e.g., `log()`) and existing studies [37, 55, 56]. Our results show that GRIST detects 78 bugs within the given time limit (i.e., 30 minutes), among which 56 are unknown bugs (i.e., the latest commit for the corresponding DL program still contains the bug). It only misses one known bug in those programs. Through submitting them to the corresponding GitHub issue repositories, eight bugs have been confirmed and three bugs have been fixed by developers. Also, GRIST can save 8.79X execution time to expose numerical bugs compared to running the original programs with their provided inputs, and expose bugs in a much more stable fashion (76 bugs can always be triggered by GRIST in all 10 repeated runs while only 37 bugs can always be triggered by running the original programs with their provided inputs in all 10 repeated runs). Compared to the state-of-the-art technique DEBAR (which is a static technique) on the same set of DL programs, DEBAR produces 12 false positives and misses 31 true bugs (of which 30 bugs can be found by GRIST), while GRIST only misses one bug and has no false positive. The results demonstrate the superiority of GRIST.

In summary, the contributions of this work are as follows:

- We propose GRIST, the first dynamic technique to expose numerical bugs in DL programs, based on gradient back-propagation.
- We conduct an experimental study based on 63 real-world DL programs. GRIST finds 78 bugs from these programs and misses only one known bug. It outperforms a simple strategy of running these programs with their provided inputs (and hoping to trigger numerical exceptions) and a state-of-the-art static technique DEBAR.
- We release our tool and dataset containing 79 real-world numerical bugs in DL programs, which can be found at: **https://github.com/Jacob-yen/GRIST**.

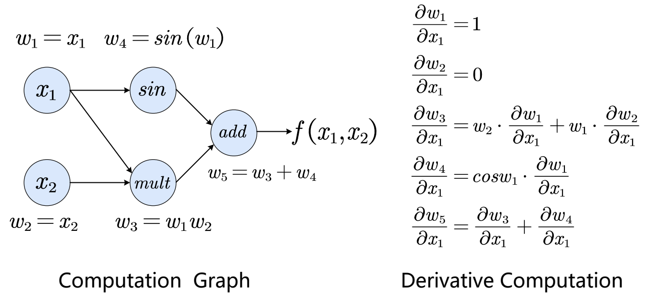

$w_1 = x_1$
$w_4 = sin(w_1)$
x_1
sin
add
$f(x_1,x_2)$
x_2
$mult$
$w_5 = w_3 + w_4$
$w_2 = x_2$
$w_3 = w_1 w_2$
$\frac{\partial w_1}{\partial x_1} = 1$
$\frac{\partial w_2}{\partial x_1} = 0$
$\frac{\partial w_3}{\partial x_1} = w_2 \cdot \frac{\partial w_1}{\partial x_1} + w_1 \cdot \frac{\partial w_2}{\partial x_1}$
$\frac{\partial w_4}{\partial x_1} = cosw_1 \cdot \frac{\partial w_1}{\partial x_1}$
$\frac{\partial w_5}{\partial x_1} = \frac{\partial w_3}{\partial x_1} + \frac{\partial w_4}{\partial x_1}$
Computation Graph
Derivative Computation

Figure 3: Computation graph and derivatives computation for $f(x_1, x_2) = x_1x_2 + sin(x_1)$ using Automatic Differentiation

2 BACKGROUND AND CHALLENGES

2.1 Gradient Computation in Deep Learning via Automatic Differentiation

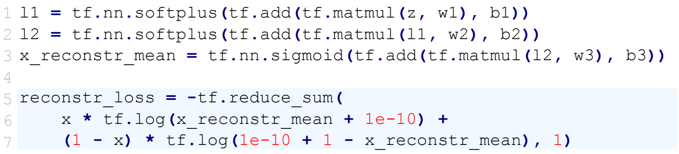
Automatic differentiation (AD) [42] is a technique that can compute the derivative of a runtime value (during program execution) over a given (input) variable, denoting the level of sensitivity of the value to the variable. Assume the runtime value is a function $f(x)$ of the input variable x. Mathematically, the derivative is defined as follows.

$$f'(x) = \lim_{\epsilon \longrightarrow 0} \frac{f(x+\epsilon) - f(x)}{\epsilon}$$

Directly computing derivatives based on the above formula is difficult for a program, which is discrete by nature. AD decomposes the function into a sequence of elementary arithmetic operations such as +, −, ×, ÷, *log*, *cos*, and *sin*, which can be automatically done by tracking the runtime data flow of individual statements in the program. By repeatedly applying the *chain rule of derivative computation* [31] to these operations, the derivative of the whole function can be automatically calculated. Figure 3 shows a simple computation graph for a function $f(x_1, x_2) = x_1x_2 + sin(x_1)$ and the corresponding derivative computation $\frac{\partial f(x_1,x_2)}{\partial x_1}$. Observe that AD decomposes the function into simple operations and computes the derivative in a forward fashion (following the data-flow direction of the computation graph). DL frameworks such as TensorFlow and PyTorch have built-in AD support, which is used to compute gradients. Please note that in AD, we need to inform about the variable(s) over which the derivatives are computed. In DL, if we need to compute gradients/derivatives regarding a variable, we need to set the property `require_grad=True` for that variable to make it to be *trainable*. As such, the framework automatically computes the gradients for each value encountered at runtime over the trainable variables. While in DL training, model weight values are by-default set to trainable and the runtime value for which gradients are queried is the cross-entropy loss value, the mechanism is general, meaning that we can declare any variable to trainable and query the gradient of any runtime value regarding a trainable variable.

2.2 Challenges

Due to the characteristics of DL programs, exposing numerical bugs in DL programs faces the following main challenges:



```
l1 = tf.nn.softplus(tf.add(tf.matmul(z, w1), b1))
l2 = tf.nn.softplus(tf.add(tf.matmul(l1, w2), b2))
x_reconstr_mean = tf.nn.sigmoid(tf.add(tf.matmul(l2, w3), b3))

reconstr_loss = -tf.reduce_sum(
    x * tf.log(x_reconstr_mean + 1e-10) +
    (1 - x) * tf.log(1e-10 + 1 - x_reconstr_mean), 1)
```

Figure 4: Example of failing to avoid the numerical bug by adding a perturbation (ID: 35a in Table 2)

Non-determinism: The computation in DL programs has substantial non-determinism due to the natural randomness in (training) inputs, the heavy use of random numbers, and computation environment uncertainty. The natural variations in training data are inevitable. Depending on the training inputs, a numerical bug may or may not manifest itself. Random values are heavily used in the numerical computation of DL programs such as initialization, regularization, and optimization, leading to substantial non-determinism. While DL program developers may reduce randomness by fixing random seeds, this may lead to degradation of model accuracy and robustness. In fact, a popular way to improve robustness is to introduce more randomness during training [32, 50]. In addition, the exposure of numerical bugs may also be affected by runtime environment such as GPU [5]. Due to the inherent non-determinism, numerical bugs may not be exposed before release, which could amplify the damage. On one hand, other users may adopt the buggy DL program to build DL models based on their own training data, and then the numerical bugs may manifest themselves. On the other hand, the numerical bugs that manifest in real system usage tend to be more devastating since it could cause unexpected system behaviors, even crash the system. Moreover, due to non-determinism, it is challenging to reproduce numerical bugs, which could largely aggravate debugging difficulty. In fact, we have found in many TensorFlow GitHub issues and PyTorch Forum posts, developers complained that they cannot reproduce the numerical bugs reported by users.

Lengthy Training: DL programs typically require lengthy training (which is dominated by numerical computation) with a large amount of data, in order to achieve high model accuracy. The typical training time of DL programs ranges from a few minutes to several days. As such, a numerical bug may only manifest itself after hours or even days into the training process. Since it is often necessary to repeat the training process of a DL program several times during the process of identifying the root cause of a numerical bug and validating fix(es), debugging may be prohibitively expensive and quick failure induction is critical.

Complexity: Due to the heavy and complex numerical computation in DL programs, numerical exceptions may have lengthy and subtle failure-inducing chains, making diagnosis difficult. Specifically, numerical bugs are difficult to find during code review since they are often caused by complex component interactions [56]. Even though simple checks/perturbations can be added to operations with the goal of avoiding numerical bugs, e.g., adding a small value ϵ to a non-negative variable x in $log(x)$ operations (to avoid $log(0)$ exceptions), they may change program semantics and degrade readability. In many cases, such checks are redundant in a broader view because the preconditions may already preclude the invalid values. Even worse, these safety checks and perturbations may be implemented incorrectly. For example, as shown in Figure 4, `1e-10` is added to the parameters in the *log* operations in order to avoid the occurrences of $log(0)$, which is a common trick by DL developers. However, in this case, due to the specific floating-point precision of the host machine, 1 derived from floating-point computation is represented as a number that is larger than 1 + *1e-8* but smaller than 1 + *1e-7*. Thus, when `x_reconstr_mean` holds the representation of value 1, $\mathtt{1e(-10) + 1 - x\_reconstr\_mean}$ yields a value smaller than zero in the second $log()$ operation, leading to an NaN. Another PyTorch example is shown in Figure 2b presented in Section 1. Although a small value `self.eps` has been added to the denominator at Line 8, a numerical bug still occurs since the derivative of `sigma.sqrt()` is accessed when `sigma` is zero and `sqrt` has no derivative at zero.

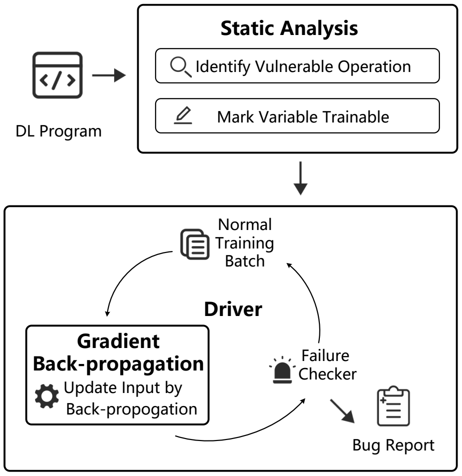


Figure 5: Overview of GRIST

3 APPROACH

3.1 Overview

To efficiently and effectively expose numerical bugs in DL programs, we develop an automated technique called GRIST. It aims to help DL-program developers or users to generate failure-inducing inputs, which include training samples and external values (e.g., those produced by random number generators). Bugs triggered by random values are as important as those triggered by training samples since if there exist certain random values that could trigger numerical bugs (e.g., NaN or INF), even though such bugs may not manifest themselves most of the time, they are latent and could be triggered some time in the future. In particular, a numerical bug manifested after system release is even more devastating since it could cause unexpected behaviors during real system usage [56].

As shown in Figure 5, our technique GRIST consists of three main components: ① *static analysis component*, ② *gradient back-propagation component*, and ③ *driver*. Given a DL program, the static analysis component analyzes the program to identify: (1) the operations that are susceptible to numerical exceptions such as `log(x)` operation that is not guarded by a (>0) range check (note that an operation without such check may not be a real bug, indicating that identifying this condition alone is not accurate enough for bug finding) and (2) the external values. As such, we mark the variables denoting external values as *trainable* such that TensorFlow and PyTorch will track gradients of these variables at runtime. A loss function called *suspect loss* is then constructed for each vulnerable operation. Intuitively, the loss function describes the distance (from the current variable value) to an invalid value that can expose a specific numerical bug. Minimizing the loss function by changing the external values through gradient back-propagation is essential to push the value at the suspect operation to become invalid.

The gradient back-propagation component updates external values based on two strategies. Please note that in the remainder of the paper, we use the terms *external values* and *inputs* interchangeably. The first one is for *iterative inputs*, which are inputs that impact program states through multiple iterations. Training samples and random weight perturbations are iterative inputs as they affect the model execution states cumulatively through many steps. In particular, for each suspect (operation), GRIST identifies all the external values whose gradients with respect to the suspect loss of the operation are non-zero, suggesting that these values have data flow reaching the suspect. GRIST updates their values along the opposite direction of the gradient sign with a constant delta. Intuitively, this is similar to how inputs are mutated in *adversarial sample generation* [35]. The difference lies in that adversarial sample generation updates a single sample input based on a cross-entropy loss or a logits loss of the output, while GRIST updates any external values that are related to some internal operation susceptible to numerical bugs.

The second kind of inputs is *non-iterative*, meaning that they contribute to the program state once (when they are loaded). Random initializations that do not happen iteratively belong to this category. For these inputs, GRIST does not update them iteratively. Instead, GRIST approximates the relation between the suspect operation and an external value with a linear function that can be derived from the gradient, and then directly infers a new value that can induce an invalid value at the suspect operation. Intuitively, since the complexity of the correlation between the external value and the suspect operation is not growing with the iteration number, there is a good chance we can approximate it with a relatively simple function and directly derive the failure-inducing value, achieving cost-effectiveness. The two kinds of inputs are distinguished by their loading places.

The driver component is responsible to update the training batch and/or restart the execution if needed so that the external value changes (made by the gradient back-propagation component) can take effect. Intuitively, at the end of each training iteration, it updates the training batch by replacing only a small number of samples that are not important (for inducing bugs at the suspect operation) with new samples. In other words, it retains those that are important (and hence must have gone through non-trivial changes by gradient back-propagation). Fresh samples are needed to prevent the failure-inducing input generation process from being trapped in some local optima (that cannot trigger the numerical bug).

If a numerical bug can be triggered within a time limit, the buggy operation and the corresponding failure-inducing external value(s) are reported. In the following, we will explain the details of each individual component.

Table 1: Vulnerable operations

Operation	Valid Range	Error Type
Division(y, x)	$x \neq 0$	Invalid value
Exp(x)	$x < 88$	
Expm1(x)	$x < 88$	
Log1p(x)	$x + 1 > 0$	
Log(x)	$x > 0$	
Sqrt(x)	$x >= 0$	
Lgamma(x)	$x \neq k$ $k \in \{0, -1, -2, -3, \dots -inf\}$	
Sqrt(x)	$x > 0$	Invalid derivative
Acos(x)	$-1 < x < 1$	

3.2 Static Analysis to Identify Vulnerable Operations and External Values

Intuitively, the essence of GRIST is no different from that of the large body of existing software testing techniques, which is to identify and model causality between some inputs and a possible failure program point, and then derive the input values that can trigger the failure. While existing techniques leverage static, dynamic, and/or symbolic analysis to derive such causality, GRIST piggy-backs on the underlying gradient computation mechanism of DL development infrastructures. As mentioned in Section 2.1, when a variable is declared trainable, the underlying infrastructure will compute its gradient for any runtime value, denoting how sensitive the runtime value is to the variable's value change. If there are multiple trainable variables, a matrix of gradients is computed for any runtime value regarding all these variables. If there is no data flow between a runtime value and a trainable variable, the corresponding gradient must be 0. As such, the static analysis essentially identifies all the possible starting points (i.e., external values) and all the possible end points of causality (i.e., operations vulnerable to numerical bugs). GRIST then marks the starting points as trainable and observes at an end point if any of the trainable variables have non-zero gradient at this point. If so, GRIST will use gradient back-propagation to change the variable(s), trying to induce failure. Examples can be found later in the section.

Vulnerable Operations. Following the existing work [56], we consider a list of vulnerable operations shown in Table 1 in our work. This is because as investigated by the existing work [56], these operations are the most frequent and have a high possibility to cause numerical bugs. For example, `exp()` may cause NaN or INF when its input is greater than 88 because of overflow. Please note that some operations may implicitly trigger numerical bugs and their invalid ranges are not very obvious. That is, there are several operations that trigger numerical bugs due to undefined derivatives as shown in Table 1. For example, although -1 and 1 are valid for `acos()`, numerical bugs still happen when the DL program tries to obtain the derivative of `acos()` at -1 or 1. GRIST identifies all the occurrences of these operations in the DL program that do not have explicit range checks as those shown in Table 1.

Defining Suspect Loss. For a vulnerable operation $T(x)$, GRIST constructs its suspect loss automatically according to its valid ranges.

In the simplest scenario, let $T(x)$ have a valid input range $x > c$, GRIST constructs its suspect loss $f(i) = x_i - c$. Here i denotes the external input (and hence the loss is a function of the input) and x_i denotes that the operand/parameter x at operation T is a function of i. As such, any update to the external value i that reduces $f(i)$ is heading towards inducing a failure at the operation. For $T(x)$ with multiple valid ranges, denoted as $(l_1, u_1) \cup (l_2, u_2) \cup ... \cup (l_k, u_k)$ without losing generality, GRIST constructs a loss function for each of the boundary values as follows.

$$f_{l_t}(i) = x_i - l_t$$

$$f_{u_t}(i) = u_t - x_i, \text{with } t \in [1, k]$$

At runtime, let $x_i \in (l_t, u_t)$, GRIST uses $f_{l_t}(i)$ if $x_i - l_t < u_t - x_i$, $f_{u_t}(i)$ otherwise. Take `Lgamma()` as an example (the logarithm of the absolute value of gamma function). In the implementation of TensorFlow and PyTorch, its valid range is that $x \neq k$, with $k \in \{0, -1, -2, -3, ... -inf\}$. Assume x_i belongs to (-5,-4) and $x_i - (-5) < -4 - x_i$, we use $f(i) = x_i - (-5)$.

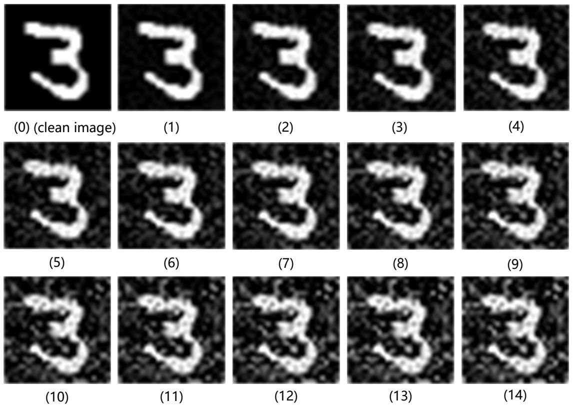
Currently, GRIST considers one vulnerable operation at a time. In other words, it uses the suspect loss function for one operation in input mutation. Since the average number of vulnerable operations in a program is usually not large, our design is reasonable. Considering multiple vulnerable operations at the same time entails using multiple suspect loss functions, whose optimization directions may be contradictory, rendering ineffectiveness.

External Values. We currently consider the following two kinds of external values: *training inputs* and *values generated by random number generators.* GRIST marks them as trainable in order to compute gradients. For training inputs, similar to adversarial sample generation, GRIST marks the input vectors after being loaded from the input file and preprocessed as trainable. For random values, GRIST marks the variables that hold the return values of random number generators as trainable.

Example. Figure 6a presents a simplified buggy code snippet from a GitHub DL program for MNIST [2]. The training loop is in Lines 13-16, in which a `cross_entropy` loss is computed. Lines 1-2 specify the input and output vectors. Line 6 denotes the computation of a hidden layer, followed by max-pooling at Line 7. Softmax is applied at Line 10 and `cross_entropy` is computed at Line 11. Our static analysis identifies that the `log()` operation at Line 11 is a vulnerable operation (as it does not have any range check), and Lines 1-2 denote iterative inputs as they are repeatedly loaded in the training loop. Please note that the ground truth label vector `y_` is also input in our context as it is loaded from some external file. As such, vectors `x` and `y_` at Lines 1 and 2 are possible starting point (of a failure causal path) and marked trainable; and `y_conv` at Line 11 is a possible end point from which GRIST constructs the suspect loss. Please note that some statements between Line 7 and Line 8 were omitted due to the space limit and the complete code (including complete data/control dependency between `x` and `y_conv`) can be found at [2]. In this case, since the parameter of a log operation ought to be greater than 0, the suspect loss is $f(x) = y\_conv - 0$ regarding the starting point of `x`. Our goal is hence to change `x` such that `y_conv` becomes smaller-than or equal-to 0.

```
1  x = tf.placeholder(tf.float32, shape=[None, 784])
2  y_ = tf.placeholder(tf.float32, shape=[None, 10])
3  x_image = tf.reshape(x, [-1, 28, 28, 1])
4  W_conv1 = weight_variable([5, 5, 1, 32])
5  b_conv1 = bias_variable([32])
6  h_conv1 = tf.nn.relu(conv2d(x_image, W_conv1)+b_conv1)
7  h_pool1 = max_pool_2x2(h_conv1)
   # omit some internal statements
8  W_fc2 = weight_variable([1024, 10])
9  b_fc2 = bias_variable([10])
10 y_conv = tf.nn.softmax(tf.matmul(h_fc1_drop,W_fc2)+b_fc2)
11 cross_entropy = -tf.reduce_sum(y_*tf.log(y_conv))
   # omit some internal statements
12 mnist = input_data.read_data_sets("data",one_hot=True)
13 for i in range(20000):
14     batch = mnist.train.next_batch(50)
15     feed_dict={x: batch[0], y_: batch[1], keep_prob: 1.0}
16     loss,_ = sess.run([cross_entropy, train_step],feed_dict)
```

(a) Simplified buggy code snippet



(b) Mutated images

Figure 6: Example of gradient back-propagation for iterative inputs from [2]

3.3 Gradient Back-Propagation

Back-propagation for Iterative Inputs. Assume the suspect loss at a vulnerable operation is $f(i)$ with i a vector of external inputs. GRIST updates i at the end of the t^{th} iteration as follows, with i_t denoting the i value at t.

$$\Delta i = \epsilon \times sign(\nabla f(i_t)) \tag{1}$$

$$i_{t+1} = clip\ (i_t - \Delta i, min, max) \tag{2}$$

In the formula, *sign* returns the sign of a real number and ϵ is a hyperparameter that determines how fast GRIST updates the input. The formula means that GRIST acquires the gradient sign of the suspect loss and updates the input by ϵ along the opposite direction of gradient sign. The updated input value needs to be clipped to its legal range.

Example. Consider the example in Figure 6a again. Although our static analysis marks both `x` and `y_` in Lines 1 and 2 to trainable respectively, at runtime GRIST observes that the gradient of `y_` is 0 at `y_conv` at Line 11, indicating `y_` does not affect the parameter of

```
 3 # define inputs and weights above
 4 gain = tf.get_variable(name="G_matmul_1",
 5                        initializer=tf.random_uniform([n_hidden],
 6                        minval=0, maxval=16))
 7
 8 # skip intermediate calculation
 9 #....
10
11 curr_scale = tf.multiply(max_scale, S)
12 new_scale = tf.div(curr_scale, gain)
```

Figure 7: Example of gradient back-propagation for non-iterative inputs from [3]

the `log()` operation. As such, it focuses on changing the value of `x`. Since it is a vector and the individual elements of the vector denote image pixels and may have different gradients, these pixels undergo different scales of mutation. The images in Figure 6b demonstrate these mutations. Observe that unlike adversarial sample generation, we do not need to bound the mutation to some norm.

Back-propagation for Non-iterative Inputs. Assume the suspect loss is $f(i)$ at some operation with i a vector of non-iterative inputs. We approximate $f(i)$ with a linear function, particularly $f(i) = g_i \times i + b$, with $g_i = \nabla f(i)$ the gradient of the suspect loss over input i computed by the infrastructure. Assume τ is an invalid value we want to reach at the suspect operation. GRIST can directly update the input i as follows.

$$\Delta i = \frac{f(i) - \tau}{\nabla f(i)} \tag{3}$$

$$i' = clip\,(i - \Delta i, min, max) \tag{4}$$

Intuitively, it solves the aforementioned linear function to make it achieve the invalid value τ. Please note that τ can be easily derived from the valid range of the operation parameter (Table 1). This strategy is very effective in practice as non-iterative inputs tend to be used in low complexity computation that can be sufficiently approximated by a linear function. Note that a simple non-linear functions can be easily approximated by multiple linear functions. For cases where linear updates cannot trigger a bug within a small number of rounds, GRIST resorts to gradient sign based mutations like for iterative inputs.

Example. Consider another example in Figure 7. It is from a Stochastic Computing Deep Neural Network (SCDNN) program for MNIST in GitHub [3]. In this case, the static analysis identifies the `div()` operation at Line 12 is vulnerable to an invalid divisor value of 0 and variable `gain` at Line 4 is a non-iterative input as it is used in the initialization phase. The variable is marked trainable. At runtime, GRIST identifies that the input variable `gain` has a non-zero gradient (i.e., gradient is equal to 1) at the divisor at Line 12 as the variable is directly used as the divisor. GRIST approximates the relation between `gain` and the divisor with a linear function $f(gain) = 1 \times gain$. According to Formula (3), Δ`gain`= `gain` and the variable is updated to 0 in the next execution according to Formula (4), triggering an NaN value.

3.4 Driver

The driver is responsible to include the mutated inputs in model execution so that the mutation can take effect and lead to failures. If the inputs being updated are non-iterative, the driver simply restarts the execution with the updated inputs. In the following, we focus on discussing how the driver handles iterative input updates. We cannot directly use the default training batching algorithm, which tends to use different inputs for each iteration. As such, the mutated inputs have no impact. A simple strategy would be to restart every time after update. However, the boot-up process is very expensive. As such, our driver tends to retain all the important inputs (i.e., the inputs that have strong causality with the numerical bug to trigger) and replaces the non-important ones with fresh samples, in order to avoid being stuck in local optima. In particular, we compute an importance score for each input i at the end of t^{th} iteration as follows.

$$score_t = \frac{update_t - clip_t + 1}{m + 10^{-7}} \tag{5}$$

In this formula, m refers to the number of iterations that input i has been updated among t iterations. Please note that m should be less than t as input i may be added during training. And $update_t = \sum_1^m u_k$ where u_k refers to the ratio of the number of elements (e.g., pixels of an image) of input i updated in the k^{th} iteration to the total number of elements in the input, $clip_t = \sum_1^m c_k$ where c_k refers to the ratio of the number of elements beyond its legal range after input i is updated in the k^{th} iteration to the number of elements. A high score indicates that the input contributes more to expose the numerical bug. At the end of each iteration, the driver replaces 5% (called the *switch rate*, a hyper-parameter in GRIST) inputs that have the lowest scores with fresh ones. Please note that $update_t$, $clip_t$, and m are 0 for newly added inputs, which hence have the highest scores among all the inputs.

Termination Condition. Termination condition determines when GRIST should give up on a suspect. We currently have a simple termination condition. We use both a fixed time limit (*timeout*) and the trend of loss function[36, 41].

4 EVALUATION

In this section, we aim to address the following research questions:

- RQ1: Is GRIST effective for exposing numerical bugs in DL programs?
- RQ2: How does GRIST perform compared with the state-of-the-art technique DEBAR?
- RQ3: Does our data replacement strategy in the driver improve the effectiveness of GRIST?

Experimental Datasets: In our study, we consider both TensorFlow programs and PyTorch programs since they are two most widely-used DL frameworks and involve both static computation graphs and dynamic computation graphs. In total, we collected 63 DL programs with 79 numerical bugs (each DL program contains at least one numerical bugs) as subjects from the following two sources: (1) *Known bugs from existing studies and GitHub:* We used 17 subjects containing 23 known bugs from existing studies and GitHub. Specifically, we used eight subjects from the existing empirical study on TensorFlow program bugs [55] and one subject from

TensorFuzz [37] following the existing work [56]. Regarding known bugs from GitHub, we adopted bug-relevant keywords (including NaN, INF, and the operations listed in Table 1) to search a set of candidate programs from GitHub according to the descending order of GitHub searching relevance and then conducted manual filtering. Since different DL programs tend to require different runtime experiments, dependencies, and datasets, it is non-trivial to run a DL program and reproduce its bugs successfully. Therefore, we used eight subjects whose 10 bugs can be reproduced conveniently and successfully in our runtime experiment. (2) *Unknown bugs from GitHub:* We applied GRIST and the state-of-the-art technique DEBAR to fuzz GitHub DL programs and finally identified 46 subjects with 56 unknown numerical bugs to developers. Specifically, we first collected a set of GitHub DL programs, each of which contains at least one operation listed in Table 1 and can run successfully in our runtime experiment, according to the descending order of GitHub searching relevance with the considered vulnerable operations. Then, we applied GRIST and the state-of-the-art technique DEBAR to the latest commit of each program, respectively. If at least one technique can detect a numerical bug within 60 minutes in a DL program, we regarded this DL program as a subject.

In particular, we consider the diversity of our subjects. Besides different DL frameworks and different types of computation graphs, our subjects also include different neural network architectures (e.g., CNN, RNN, and GAN) and different datasets (e.g., MNIST, Fashion-MNIST, and User-defined Data).

Experimental Settings: To answer RQ1, we ran each subject with and without GRIST using its default dataset and hyperparameters. GRIST has 3 hyperparameters: *timeout* (the time limit for running GRIST), ϵ that defines the input update rate, and *switch rate* that specifies the fraction of samples that are replaced at each batch for iterative inputs. Specifically, *timeout* is set to 30 minutes; ϵ is set to 0.15; and *switch rate* is set to 5%. We have investigated the influence of main parameters in Section 5.1. Note that with GRIST, the inputs are mutated during execution. To mitigate non-determinism (e.g., numerical exceptions being randomly triggered), we repeated each run 10 times and reported the aggregated results. To answer RQ2, we applied DEBAR with its default hyperparameters to each subject and also set its time limit to 30 minutes for fair comparison. To answer RQ3, we ran each subject through GRIST without its data replacement strategy, while the other two hyperparameters in GRIST remain the same.

Hardware and Runtime Environments: Our experiment was conducted on the Intel Xeon Silver 4214 machine with 128GB RAM, Ubuntu 16.04.6 LTS, and two GTX 2080 Ti GPUs. We used the Anaconda environments to switch different versions of PyTorch and TensorFlow.

4.1 RQ1: Overall Effectiveness of GRIST

Setup. We ran 63 subjects containing 79 bugs 10 times with and without GRIST, respectively. Table 2 shows the comparison results between with and without GRIST, in which C is the total number of times that a bug is exposed in 10 repeated runs, T refers to the average execution time for exposing a bug. Please noted that if a numerical bug is exposed in 3 out of the 10 runs, only the time in these 3 times are used to calculate the average result. We calculated the average improvement achieved by GRIST in terms of the execution time for each bug (denoted as $\Uparrow T$). We also calculated the average results for the overall 79 bugs as shown in the last row in Table 2. For those bugs that were not triggered within the given time limit, we used the time limit (i.e., 30 minutes) to calculate the overall average time. Due to the space limit, we use ID to replace the subject name, and the complete information about our subjects can be found at our project homepage[1].

Results. Table 2 shows the effectiveness of GRIST in exposing numerical bugs and accelerating failure triggering. Overall, GRIST is able to successfully detect 78 (out of 79) bugs within 30 minutes, among which 56 are unknown bugs (i.e., the latest commit for the corresponding subject still contains the bug). In particular, 26 of 56 unknown bugs cannot be detected by the state-of-the-art technique DEBAR, demonstrating the unique superiority of GRIST (more detailed comparison with DEBAR can be found in Section 4.2). Through submitting them to the corresponding issue repositories and communicating with developers, eight bugs have been confirmed and three bugs have been fixed. We further analyzed the bug that was not exposed by GRIST (i.e., ID: 17, which cannot be triggered by running the original program with the default inputs either) and found that GRIST indeed pushes the parameter value of the vulnerable operation (i.e., *exp*) very close to the boundary but cannot go beyond (to trigger the failure). By relaxing the time limit to one hour, GRIST is able to trigger the bug (with average time of 58 minutes).

From Table 2, there are 34 bugs, which were never be exposed in the 10 runs of using the default inputs. In contrast, GRIST can always trigger 76 numerical bugs in all the 10 runs and the remaining two bugs in some of the 10 runs (due to inherent non-determinism). Regarding the 45 bugs that can be exposed by both GRIST and default inputs, GRIST can trigger them in a much more stable fashion. Specifically, GRIST can trigger them in all the 10 runs whereas using the default inputs triggers eight of them in some of the 10 runs (even less than 5 times for the subject with ID-24). Also observe from Table 2 that GRIST can substantially reduce the time spent on triggering bugs. Overall, GRIST can save 8.79X time cost on average. In particular, for the bug (ID-37), using the default input took 1,586.53 seconds to trigger it while GRIST took only 0.69 seconds, saving 2,299.32X time cost. There is only one bug (i.e., ID: 2a) that GRIST spends longer average time on triggering it than the original program with the default input. We analyzed that for this bug, using the default input alone took only 0.40 seconds to trigger it. For such a bug, GRIST cannot accelerate the process that is already extremely fast.

4.2 RQ2: Comparison with the State-of-the-Art Technique DEBAR

Setup. For comparison with the state-of-the-art technique DEBAR, we applied it to each subject and used ✓/✗ to mark whether DEBAR can detect the bug or not in Table 2. As DEBAR does not need to run programs, we do not need to run it 10 times.

Results. As expected, the execution time of the static technique DEBAR is only 2 seconds on average across all the subjects, but

[1] https://github.com/Jacob-yen/GRIST.

Table 2: Results for using the default inputs,GRIST,GRIST$_{NS}$ and DEBAR

ID	Default Input		GRIST			GRIST$_{NS}$			DEBAR	ID	Default Input		GRIST			GRIST$_{NS}$			DEBAR
	C	T	C	T	⇑ T	C	T	⇑ T			C	T	C	T	⇑ T	C	T	⇑ T	
1	10	814.86	10	16.12	50.55 X	10	14.30	57.00 X	✓	32	10	11.12	10	0.08	139.00 X	10	0.08	139.00 X	✗
2a	10	0.40	10	16.70	-41.92 X	10	8.63	-21.66 X	✓	33	10	11.74	10	0.28	41.93 X	10	0.28	41.93 X	✗
2b	10	0.32	10	0.31	1.03 X	10	0.25	1.28 X	✓	34	10	130.72	10	0.22	594.18 X	10	0.22	594.18 X	✗
3	10	24.66	10	7.21	3.42 X	10	25.17	-1.02 X	✓	35a	0	—	10	14.65	+∞	10	12.00	+∞	✓
4	0	—	10	0.43	+∞	10	0.41	+∞	✗	35b	0	—	10	308.64	+∞	10	353.27	+∞	✗
5	0	—	10	0.34	+∞	10	0.34	+∞	✗	36a	7	999.70	10	15.17	65.9 X	10	9.10	109.82 X	✓
6	6	1,451.17	10	19.85	73.11 X	10	10.21	142.07 X	✓	36b	0	—	10	306.53	+∞	10	352.52	+∞	✗
7	8	1,464.85	10	19.84	73.83 X	10	10.25	142.91 X	✓	37	10	1,586.53	10	0.69	2299.32 X	10	0.69	2299.32 X	✗
8	8	1,461.66	10	19.76	73.98 X	10	10.27	142.34 X	✓	38	0	—	10	0.28	+∞	10	0.28	+∞	✗
9a	10	57.00	10	5.30	10.75 X	10	4.00	14.25 X	✓	39a	8	718.60	10	12.02	59.78 X	10	8.35	86.01 X	✓
9b	10	61.41	10	19.96	3.08 X	10	17.78	3.45 X	✓	39b	0	—	10	309.12	+∞	10	350.39	+∞	✗
10	10	383.03	10	43.86	8.73 X	10	220.84	1.73 X	✓	40	10	547.80	10	87.30	6.27 X	10	155.40	3.53 X	✗
11a	10	510.92	10	5.93	86.16 X	10	4.28	119.37 X	✓	41	10	556.30	10	87.00	6.39 X	10	180.40	3.08 X	✗
11b	10	556.37	10	5.37	103.61 X	10	3.95	140.85 X	✓	42	10	548.90	10	85.90	6.39 X	10	120.50	4.56 X	✗
11c	0	—	10	4.58	+∞	10	4.97	+∞	✓	43a	0	—	10	13.25	+∞	10	10.73	+∞	✓
12	9	220.25	10	52.06	4.23 X	10	133.12	1.65 X	✗	43b	0	—	10	308.29	+∞	10	353.90	+∞	✗
13	0	—	10	0.65	+∞	10	0.52	+∞	✗	44	0	—	10	231.20	+∞	10	208.72	+∞	✓
14	10	564.72	10	86.23	6.55 X	9	336.08	1.68 X	✓	45a	10	262.36	10	90.86	2.89 X	0	—	−∞	✓
15	10	700.90	10	14.96	46.86 X	10	13.00	53.91 X	✓	45b	10	1,278.81	10	27.46	46.57 X	10	41.54	30.79 X	✓
16a	0	—	10	3.30	+∞	10	68.74	+∞	✓	46	0	—	10	0.21	+∞	10	0.21	+∞	✗
16b	10	534.36	10	3.29	162.42 X	10	5.24	101.98 X	✓	47	0	—	10	0.19	+∞	10	0.19	+∞	✗
16c	0	—	10	4.43	+∞	10	4.89	+∞	✓	48a	10	40.50	10	0.94	43.09 X	10	1.19	34.03 X	✓
17	0	—	0	—	—	0	—	—	✗	48b	10	440.12	10	0.84	523.95 X	10	1.00	440.12 X	✓
18	10	343.66	10	25.49	13.48 X	5	549.64	-1.60 X	✓	49a	0	—	10	13.39	+∞	10	10.34	+∞	✓
19	10	855.71	10	137.31	6.23 X	0	—	−∞	✓	49b	0	—	10	307.24	+∞	10	351.98	+∞	✗
20	0	—	10	608.80	+∞	0	—	—	✓	50	0	—	10	166.00	+∞	0	—	—	✓
21	0	—	10	44.49	+∞	0	—	—	✓	51	0	—	3	1,520.18	+∞	0	—	—	✗
22	0	—	10	1,119.60	+∞	0	—	—	✗	52	10	404.30	10	61.50	6.57 X	0	—	−∞	✓
23	0	—	10	0.21	+∞	10	0.21	+∞	✗	53	0	—	10	0.27	+∞	10	0.27	+∞	✗
24	4	1,639.14	10	41.32	39.67 X	10	72.52	22.60 X	✓	54	10	3.25	10	0.13	25.00 X	10	0.13	25.00 X	✗
25	10	502.05	10	59.36	8.46 X	0	—	−∞	✓	55	10	1,322.20	10	32.70	40.43 X	10	37.03	35.71 X	✓
26	10	16.95	10	0.27	62.78 X	10	0.27	62.78 X	✗	56	10	11.15	10	0.20	55.75 X	10	0.20	55.75 X	✗
27	0	—	10	0.19	+∞	10	0.19	+∞	✗	57	0	—	10	0.25	+∞	10	0.25	+∞	✗
28a	0	—	10	1.11	+∞	10	1.30	+∞	✓	58	10	1,283.60	10	40.70	31.54 X	10	15.26	84.12 X	✓
28b	0	—	10	176.02	+∞	10	176.02	+∞	✓	59	10	167.50	10	0.10	1675. X	10	0.09	1861.11 X	✓
28c	0	—	10	176.02	+∞	10	176.02	+∞	✓	60	10	131.10	10	41.80	3.14 X	0	—	−∞	✓
28d	0	—	10	626.12	+∞	10	626.12	+∞	✓	61	10	579.60	10	27.00	21.47 X	6	413.92	1.40 X	✓
29	9	852.69	10	44.02	19.37 X	10	81.79	10.43 X	✓	62	10	839.80	10	155.40	5.4 X	7	451.66	1.86 X	✗
30	10	133.46	10	45.78	2.92 X	0	—	−∞	✓	63	0	—	10	256.00	+∞	7	1,207.52	+∞	✗
31	0	—	3	23.79	+∞	1	18.82	+∞	✓	**Total**	**429**	**1,091.47**	**766**	**124.11**	**8.79**	**645**	**365.19**	**2.99**	**48(✓)/31(✗)**

[*] +∞ means that the GRIST or GRIST$_{NS}$ based run(s) can expose numerical bugs in the 10 runs while the default inputs cannot; −∞ means GRIST$_{NS}$ cannot find numerical bugs in the 10 runs while the default inputs can; — indicates that the corresponding technique cannot expose the numerical bugs; ✓/✗ means that DEBAR can detect the bug or not.

indeed DEBAR reports 12 FPs (false positives), which have been extensively explained in the work proposing DEBAR [56]. Also, there are 31 (out of 79) bugs that were not detected by DEBAR, of which 30 bugs were detected by GRIST. In fact, GRIST can detect a superset of the bugs that DEBAR can detect. We manually analyzed the 31 FNs (false negatives) of DEBAR and found that there are three reasons: 1) As mentioned earlier, DEBAR cannot be applicable to dynamic computation graphs, and thus it missed to detect bugs based on dynamic computation graphs. It is remarkable that the latest version of TensorFlow has also supported dynamic computation graphs and takes it as the default usage, indicating that supporting to detect bugs based on dynamic computation graphs like GRIST will be an inevitable trend in the future to some degree. 23 of 31 FNs fall into this category. 2) DEBAR does not support the error type of invalid derivative listed in Table 1 since the derivation operation can be found only *at runtime*. 5 of 31 FNs fall into this category. 3) DEBAR requires users to manually configure the range of each primitive parameter in the program, but there are three bugs, which DEBAR cannot detect when configuring the correct range (e.g., the range of the variable after normalization is [0,1]) but can detect when setting a more coarse range (e.g., [0,inf]). The results demonstrate that GRIST outperforms DEBAR in terms of both FPs and FNs.

4.3 RQ3: Contribution of the Data Replacement Strategy in GRIST

Setup. To investigate the impact of replacing unimportant samples with fresh ones in GRIST, we prohibited GRIST from dropping inputs with low scores or adding new inputs. In other words, it continued to update the same set of samples iteratively. The settings of ϵ and $timeout$ remain the same. We call this variant GRIST$_{NS}$.

Results. Observe that replacing unimportant samples has a positive effect on the performance of GRIST. First of all, in terms of the number of exposed bugs, GRIST exposes 78 bugs in 766 runs in total while GRIST$_{NS}$ (GRIST without data replacement) only exposes 67 bugs in 645 runs. Also, there are six bugs that were not detected by GRIST$_{NS}$ but were detected by the original programs using default inputs. Second, in terms of time cost reduction in exposing bugs, GRIST$_{NS}$ can save 2.99X time cost compared to the original programs using default inputs, while that of GRIST is 8.79X.

The results demonstrate the data replacement strategy is indeed able to improve the performance of GRIST.

5 DISCUSSION

5.1 Influence of Main Parameters in GRIST

We investigated the influence of two main parameters in GRIST, i.e., ϵ (the input update rate) and *switch rate* (the fraction of samples being replaced at the end of each training iteration), by conducting an experiment based on three randomly selected subjects (ID-3, ID-18, ID-35). Regarding ϵ, we studied 0.1, 0.15, 0.2, 0.25, and 0.3, while regarding *switch rate*, we studied 0.01, 0.05, 0.1, 0.15, and 0.2, whose average results are shown in Figure 9. In the experiment, only one parameter is changed each time while others use our default settings. We found that, in general GRIST is insensitive to ϵ or *switch rate* (except 0.01) within the studied range. Regarding *switch rate* of 0.01, one subject has a small batch size such that the number of replaced data is very small, making it nearly equivalent to GRIST$_{NS}$.

5.2 Generalizability of GRIST

On one hand, GRIST can work on both static computation graphs and dynamic computation graphs, while DEBAR can only support the former, indicating that GRIST is more general than the state-of-the-art technique DEBAR for detecting numerical bugs in DL programs. On the other hand, even though GRIST is designed to expose numerical bugs in DL programs, it can be also generalized to DL libraries to some degree. This is because DL libraries can also utilize their gradient computation mechanisms that GRIST piggy-backs on, through invocations from DL programs. We use an example of the PyTorch library, shown in Figure 8, to illustrate how GRIST is generalized to detect numerical bugs in DL libraries. Figure 8a shows the function `entropy` in PyTorch, which could produce an NaN when `self.rate` is 0 in `log`. GRIST can detect this numerical bug by 1) finding or creating a DL program that invokes this function (shown in Figure 8b), 2) constructing *suspect loss* by instrumenting PyTorch to return the parameter value of `log` in `entropy` along with its original returned value (Lines 3 in Figure 8b), and 3) updating the argument value of `entropy` in the DL program via gradient computation (utilizing the gradient computation mechanism in PyTorch) between *suspect loss* and the argument `rate` of `entropy` (Lines 14-18 in Figure 8b). In this way, `rate` becomes zero eventually and the numerical bug in PyTorch is exposed. Compared with DL programs, the main difference of detecting numerical bugs in DL libraries is that the logical relationship between *suspect loss* and *input* of the library function under test lies in DL libraries rather than DL programs, and thus the parameters of the buggy operations have to be returned to DL programs from DL libraries for gradient computation.

```
1 # In torch.distributions.exponential.Exponential
2 def entropy(self):
3     return 1.0 - torch.log(self.rate)
```

(a) Function in the PyTorch library

```
1  # In Instrumented Exponential
2  def entropy(self):
3      return 1.0 - torch.log(self.rate),self.rate
4
5  # In Driver of GRIST
6  rate = initialize_rate()
7  rate = clamp(rate)
8  exponential = Exponential(rate)
9
10 while NotTerminate():
11     exponential = Exponential(rate)
12     actual_results,monitored_var = exponential.entropy()
13     suspect_loss = define_suspect_loss(monitored_var)
14     grads = calculate_gradients(suspect_loss,rate)
15     NaN_Check(actual_results)
16
17     rate = update_rate_by_grads(grads)
18     rate = clamp(rate)
```

(b) Pytorch program invoking the function (driver of GRIST)

Figure 8: Example of applying GRIST to detect a numerical bug in the PyTorch library

5.3 Threats to Validity

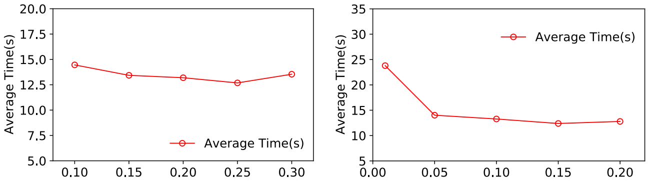
The *internal* threat to validity mainly lies in the implementation of GRIST. To reduce this threat, two authors have carefully examined the implementation of GRIST, including reviewing and testing the code. Specifically, they cross-reviewed each function and wrote unit tests. Also, regarding the integrated tool, they used the debug mode in the PyCharm IDE to ensure the correctness of the intermediate states and the final output for a program.

The *external* threat to validity mainly lies in the subjects used in our study. To reduce this threat, we collected 63 real-world DL programs containing 79 bugs from two sources as subjects in our study, including 23 known bugs from existing studies and GitHub, and 56 unknown bugs from GitHub that can be detected by either GRIST or DEBAR. Section 4 presents the subject collection process in detail. In the future, we will evaluate GRIST on more DL programs based on more DL libraries.

6 RELATED WORK

DL Program Bugs. The most related work to ours is DEBAR [56], which has been discussed and compared in Sections 2.2 and 4.2. Besides, there are a number of empirical studies on DL program bugs [8, 26–28, 53–55]. For example, Zhang et al.[55] analyzed the root causes and symptoms of 175 TensorFlow program bugs from GitHub issues and Stack Overflow posts. Humbatova et al.[26] provided a taxonomy of DL program bugs through manual analysis and interviews based on GitHub issues and Stack Overflow posts. Islam et al.[27, 28] analyzed the types, root causes, impact, and fix patterns of DL program bugs based on five popular DL libraries. Zhang et al. [54] inspected 715 questions on Stack Overflow about DL and summarized many common challenges in developing DL programs. Different from them, we focus on proposing the first dynamic technique to expose numerical bugs in DL programs.

Numerical Bugs in Traditional Software. There is some work on numerical bugs in traditional software. For example, Franco et al.[18] conducted a comprehensive study on numerical bugs in traditional software. Dietz et al.[15] developed *IOC*, a dynamic checking tool for integer overflow and conducted the first empirical study on integer overflow in C and C++ code. Tang et al.[45] proposed a toolchain that can detect potential numerical instability and diagnose the reasons for such instability. Guo et al.[22] proposed



(a) ϵ **(b)** $switch\_rate$

Figure 9: Results for different settings of ϵ **and** $switch\_rate$

an approach based on symbolic execution to efficiently generating floating-point inputs to trigger program errors. Different from them, our work targets at numerical bugs in DL programs, which are very different from traditional software as presented in Section 1.

Furthermore, Fu et al. [19, 20] adopted gradient optimization to analyze float-point code in traditional software, aiming at generating tests for high coverage. Even though they also utilized gradients, different from them, our contribution lies in handling bugs in DL programs. Specifically, DL training is extremely expensive and demands many processes, whereas the execution model of traditional numerical programs is simple. GRIST piggy-backs on existing gradient back-propagation mechanism, which makes it easily deployable. It requires solving new challenges as well such as interacting with DL primitives (e.g., automatically marking selected variables as trainable) and handling data loading.

DL Testing. Over the years, a large amount of work focus on DL testing [13, 16, 23, 29, 33, 34, 37, 39, 43, 48, 51]. However, they aim to either test DL models by proposing various input generation techniques [7, 23, 51] or designing various test criteria [25, 29, 33, 34, 39], or test DL libraries and DL compilers [40, 43, 47] Different from them, our work aims to detect DL program bugs, i.e., numerical bugs in DL programs.

7 CONCLUSION

In this paper, we propose the first dynamic technique to generate inputs to expose numerical bugs in DL programs and implement it in a tool named GRIST. The technique piggy-backs on the built-in gradient computation of the underlying deep learning framework. Our results on 63 real-world DL programs with 79 numerical bugs show that GRIST can expose unknown numerical bugs and substantially reduce the execution time needed to trigger bugs.

ACKNOWLEDGEMENT

We thank all the anonymous reviewers for their valuable comments. This work has been supported by the National Natural Science Foundation of China 62002256, 61872263, Intelligent Manufacturing Special Fund of Tianjin 20193155, IARPA TrojAI W911NF-19-S-0012, NSF 1901242, 2006688, 1910300, and ONR N000141712045, N000141410468, N000141712947. Any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflect the views of our sponsors.

REFERENCES

[1] Accessed: 2020. GitHub. https://github.com/philipperemy/deep-speaker/issues/5.
[2] Accessed: 2020. GitHub. https://github.com/ForeverZyh/TensorFlow-Program-Bugs/blob/master/StackOverflow/IPS-2/33699174-buggy/mnist.py.
[3] Accessed: 2020. GitHub. https://github.com/adamsolomou/SC-DNN/blob/a9169c6b7a0d456c1d2f229913e2d8c042c40aab/src/training/sc_train_creg.py.
[4] Accessed: 2020. PyTorch Forums. https://discuss.pytorch.org/t/my-self-implemented-batchnorm-relu-gives-nan/42294.
[5] Accessed: 2020. PyTorch Forums. https://discuss.pytorch.org/t/different-losses-on-2-different-machines/36446/5.
[6] Nathaniel Ayewah, David Hovemeyer, J. David Morgenthaler, John Penix, and William Pugh. 2008. Using Static Analysis to Find Bugs. *IEEE Softw.* 25, 5 (2008), 22–29.
[7] Houssem Ben Braiek and Foutse Khomh. 2019. DeepEvolution: A Search-Based Testing Approach for Deep Neural Networks. In *ICSME*. IEEE, 454–458.
[8] Houssem Ben Braiek and Foutse Khomh. 2019. TFCheck : A TensorFlow Library for Detecting Training Issues in Neural Network Programs. In *19th IEEE International Conference on Software Quality, Reliability and Security, QRS 2019, Sofia, Bulgaria, July 22-26, 2019*. IEEE, 426–433.
[9] Nicholas Carlini and David A. Wagner. 2017. Towards Evaluating the Robustness of Neural Networks. In *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 39–57.
[10] Chenyi Chen, Ari Seff, Alain Kornhauser, and Jianxiong Xiao. 2015. Deepdriving: Learning affordance for direct perception in autonomous driving. In *Proceedings of the IEEE International Conference on Computer Vision*. 2722–2730.
[11] Junjie Chen, Xiaoting He, Qingwei Lin, Hongyu Zhang, Dan Hao, Feng Gao, Zhangwei Xu, Yingnong Dang, and Dongmei Zhang. 2019. Continuous Incident Triage for Large-Scale Online Service Systems. In *34th IEEE/ACM International Conference on Automated Software Engineering*. 364–375.
[12] Junjie Chen, Haoyang Ma, and Lingming Zhang. 2020. Enhanced Compiler Bug Isolation via Memoized Search. In *35th IEEE/ACM International Conference on Automated Software Engineering*. 78–89.
[13] Junjie Chen, Zhuo Wu, Zan Wang, Hanmo You, Lingming Zhang, and Ming Yan. 2020. Practical Accuracy Estimation for Efficient Deep Neural Network Testing. *ACM Trans. Softw. Eng. Methodol.* 29, 4 (2020), 30:1–30:35.
[14] Junjie Chen, Shu Zhang, Xiaoting He, Qingwei Lin, Hongyu Zhang, Dan Hao, Yu Kang, Feng Gao, Zhangwei Xu, Yingnong Dang, and Dongmei Zhang. 2020. How Incidental are the Incidents? Characterizing and Prioritizing Incidents for Large-Scale Online Service Systems. In *35th IEEE/ACM International Conference on Automated Software Engineering*. 373–384.
[15] Will Dietz, Peng Li, John Regehr, and Vikram S. Adve. 2012. Understanding integer overflow in C/C++. In *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*. IEEE Computer Society, 760–770.
[16] Xiaoning Du, Xiaofei Xie, Yi Li, Lei Ma, Yang Liu, and Jianjun Zhao. 2019. A Quantitative Analysis Framework for Recurrent Neural Network. In *ASE*. IEEE, 1062–1065.
[17] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. 2013. PLDI 2002: Extended static checking for Java. *ACM SIGPLAN Notices* 48, 4S (2013), 22–33.
[18] Anthony Di Franco, Hui Guo, and Cindy Rubio-González. 2017. A comprehensive study of real-world numerical bug characteristics. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*. IEEE Computer Society, 509–519.
[19] Zhoulai Fu and Zhendong Su. 2017. Achieving high coverage for floating-point code via unconstrained programming. In *PLDI*. ACM, 306–319.
[20] Zhoulai Fu and Zhendong Su. 2019. Effective floating-point analysis via weak-distance minimization. In *PLDI*. ACM, 439–452.
[21] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. 2015. Explaining and Harnessing Adversarial Examples. In *ICLR (Poster)*.
[22] Hui Guo and Cindy Rubio-González. 2020. Efficient generation of error-inducing floating-point inputs via symbolic execution. In *ICSE*. ACM, 1261–1272.
[23] Jianmin Guo, Yu Jiang, Yue Zhao, Quan Chen, and Jiaguang Sun. 2018. DLFuzz: differential fuzzing testing of deep learning systems. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*. ACM, 739–743.
[24] Andrew Habib and Michael Pradel. 2018. How many of all bugs do we find? a study of static bug detectors. In *ASE*. ACM, 317–328.
[25] Fabrice Harel-Canada, Lingxiao Wang, Muhammad Ali Gulzar, Quanquan Gu, and Miryung Kim. 2020. Is neuron coverage a meaningful measure for testing deep neural networks?. In *ESEC/SIGSOFT FSE*. ACM, 851–862.
[26] Nargiz Humbatova, Gunel Jahangirova, Gabriele Bavota, Vincenzo Riccio, Andrea Stocco, and Paolo Tonella. 2019. Taxonomy of Real Faults in Deep Learning Systems. *CoRR* abs/1910.11015 (2019).
[27] Md Johirul Islam, Giang Nguyen, Rangeet Pan, and Alessandra Russo 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*. ACM, 510–520.
[28] Md Johirul Islam, Rangeet Pan, Giang Nguyen, and Hridesh Rajan. 2020. Repairing Deep Neural Networks: Fix Patterns and Challenges. *CoRR* abs/2005.00972 (2020).
[29] Jinhan Kim, Robert Feldt, and Shin Yoo. 2019. Guiding deep learning system testing using surprise adequacy. In *Proceedings of the 41st International Conference*

on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019. IEEE / ACM, 1039–1049.

[30] Alexey Kurakin, Ian J. Goodfellow, and Samy Bengio. 2017. Adversarial examples in the physical world. In *ICLR (Workshop)*. OpenReview.net.

[31] Ron Larson and Bruce H Edwards. 2016. *Calculus of a single variable*. Nelson Education.

[32] Xuanqing Liu, Minhao Cheng, Huan Zhang, and Cho-Jui Hsieh. 2018. Towards Robust Neural Networks via Random Self-ensemble. In *ECCV (7) (Lecture Notes in Computer Science, Vol. 11211)*. Springer, 381–397.

[33] 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*. IEEE, 614–618.

[34] 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*. ACM, 120–131.

[35] 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*. OpenReview.net.

[36] Maren Mahsereci, Lukas Balles, Christoph Lassner, and Philipp Hennig. 2017. Early stopping without a validation set. *arXiv preprint arXiv:1703.09580* (2017).

[37] 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 (Proceedings of Machine Learning Research, Vol. 97)*. PMLR, 4901–4911.

[38] Nicolas Papernot, Patrick D. McDaniel, Somesh Jha, Matt Fredrikson, Z. Berkay Celik, and Ananthram Swami. 2016. The Limitations of Deep Learning in Adversarial Settings. In *EuroS&P*. IEEE, 372–387.

[39] 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, October 28-31, 2017*. ACM, 1–18.

[40] 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.

[41] Lutz Prechelt. 1998. Early stopping-but when? In *Neural Networks: Tricks of the trade*. Springer, 55–69.

[42] Louis B. Rall. 1981. *Automatic Differentiation: Techniques and Applications*. Lecture Notes in Computer Science, Vol. 120. Springer. https://doi.org/10.1007/3-540-10861-0

[43] Qingchao Shen, Haoyang Ma, Junjie Chen, Yongqiang Tian, Shing-Chi Cheung, and Xiang Chen. 2021. A Comprehensive Study of Deep Learning Compiler Bugs. In *ESEC/FSE*. to appear.

[44] Yi Sun, Yuheng Chen, Xiaogang Wang, and Xiaoou Tang. 2014. Deep learning face representation by joint identification-verification. In *Advances in neural information processing systems*. 1988–1996.

[45] Enyi Tang, Xiangyu Zhang, Norbert Th. Müller, Zhenyu Chen, and Xuandong Li. 2017. Software Numerical Instability Detection and Diagnosis by Combining Stochastic and Infinite-Precision Testing. *IEEE Trans. Software Eng.* 43, 10 (2017), 975–994.

[46] Ferdian Thung, Lucia, David Lo, Lingxiao Jiang, Foyzur Rahman, and Premkumar T. Devanbu. 2012. To what extent could we detect field defects? an empirical study of false negatives in static bug finding tools. In *ASE*. ACM, 50–59.

[47] Zan Wang, Ming Yan, Junjie Chen, Shuang Liu, and Dongdi Zhang. 2020. Deep learning library testing via effective model generation. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 788–799.

[48] Zan Wang, Hanmo You, Junjie Chen, Yingyi Zhang, Xuyuan Dong, and Wenbin Zhang. 2021. Prioritizing Test Inputs for Deep Neural Networks via Mutation Analysis. In *43rd IEEE/ACM International Conference on Software Engineering*. 397–409.

[49] Chaowei Xiao, Jun-Yan Zhu, Bo Li, Warren He, Mingyan Liu, and Dawn Song. 2018. Spatially Transformed Adversarial Examples. In *ICLR (Poster)*. OpenReview.net.

[50] Cihang Xie, Jianyu Wang, Zhishuai Zhang, Zhou Ren, and Alan L. Yuille. 2018. Mitigating Adversarial Effects Through Randomization. In *ICLR (Poster)*. OpenReview.net.

[51] Xiaofei Xie, Lei Ma, Felix Juefei-Xu, Minhui Xue, Hongxu Chen, Yang Liu, Jianjun Zhao, Bo Li, Jianxiong Yin, and Simon See. 2019. DeepHunter: a coverage-guided fuzz testing framework for deep neural networks. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019*. ACM, 146–157.

[52] Lin Yang, Junjie Chen, Zan Wang, Weijing Wang, Jiajun Jiang, Xuyuan Dong, and Wenbin Zhang. 2021. Semi-supervised Log-based Anomaly Detection via Probabilistic Label Estimation. In *43rd IEEE/ACM International Conference on Software Engineering*. 1448–1460.

[53] Ru Zhang, Wencong Xiao, Hongyu Zhang, Yu Liu, Haoxiang Lin, and Mao Yang. [n.d.]. An Empirical Study on Program Failures of Deep Learning Jobs. ([n. d.]).

[54] Tianyi Zhang, Cuiyun Gao, Lei Ma, Michael R. Lyu, and Miryung Kim. 2019. An Empirical Study of Common Challenges in Developing Deep Learning Applications. In *30th IEEE International Symposium on Software Reliability Engineering, ISSRE 2019, Berlin, Germany, October 28-31, 2019*. IEEE, 104–115.

[55] 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*. ACM, 129–140.

[56] Yuhao Zhang, Luyao Ren, Liqian Chen, Yingfei Xiong, Shing-Chi Cheung, and Tao Xie. 2020. Detecting numerical bugs in neural network architectures. In *ESEC/SIGSOFT FSE*. ACM, 826–837.