Automatically Detecting Numerical Instability in Machine Learning Applications via Soft Assertions

SHAILA SHARMIN*, Iowa State University, USA
ANWAR HOSSAIN ZAHID*, Iowa State University, USA
SUBHANKAR BHATTACHARJEE, Iowa State University, USA
CHIAMAKA IGWILO, Iowa State University, USA
MIRYUNG KIM, University of California at Los Angeles, USA
WEI LE, Iowa State University, USA

Machine learning (ML) applications have become an integral part of our lives. ML applications extensively use floating-point computation and involve very large/small numbers; thus, maintaining the numerical stability of such complex computations remains an important challenge. Numerical bugs can lead to system crashes, incorrect output, and wasted computing resources. In this paper, we introduce a novel idea, namely *soft assertions (SA)*, to encode safety/error conditions for the places where numerical instability can occur. A soft assertion is an ML model automatically trained using the dataset obtained during unit testing of unstable functions. Given the values at the unstable function in an ML application, a soft assertion reports how to change these values in order to trigger the instability. We then use the output of soft assertions as signals to effectively mutate inputs to trigger numerical instability in ML applications. In the evaluation, we used the GRIST benchmark, a total of 79 programs, as well as 15 real-world ML applications from GitHub. We compared our tool with 5 state-of-the-art (SOTA) fuzzers. We found all the GRIST bugs and outperformed the baselines. We found 13 numerical bugs in real-world code, one of which had already been confirmed by the GitHub developers. While the baselines mostly found the bugs that report `NaN` and `INF`, our tool *Soft Assertion Fuzzer* found numerical bugs with incorrect output. We showed one case where the *Tumor Detection Model*, trained on Brain MRI images, should have predicted "tumor", but instead, it incorrectly predicted "no tumor" due to the numerical bugs. Our replication package is located at https://figshare.com/s/6528d21ccd28bea94c32.

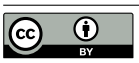
CCS Concepts: • **Software and its engineering → Software testing and debugging**; **Software verification and validation**.

Additional Key Words and Phrases: Soft Assertions, Numerical Instability, Fuzzing, Machine Learning Code

ACM Reference Format:
Shaila Sharmin, Anwar Hossain Zahid, Subhankar Bhattacharjee, Chiamaka Igwilo, Miryung Kim, and Wei Le. 2025. Automatically Detecting Numerical Instability in Machine Learning Applications via Soft Assertions. *Proc. ACM Softw. Eng.* 2, FSE, Article FSE124 (July 2025), 22 pages. https://doi.org/10.1145/3729394

*Both authors contributed equally to this research.

Authors' Contact Information: Shaila Sharmin, ssharmin@iastate.edu, Iowa State University, Ames, USA; Anwar Hossain Zahid, ahzahid@iastate.edu, Iowa State University, Ames, USA; Subhankar Bhattacharjee, Iowa State University, Ames, USA, s7bhat@iastate.edu; Chiamaka Igwilo, Iowa State University, Ames, USA, Igwilo@iastate.edu; Miryung Kim, University of California at Los Angeles, Los Angeles, USA, miryung@cs.ucla.edu; Wei Le, Iowa State University, Ames, USA, weile@iastate.edu.



ACM 2994-970X/2025/7-ARTFSE124
https://doi.org/10.1145/3729394

1 Introduction

Machine learning (ML) applications are becoming essential in a wide range of domains [13, 15, 19]. ML code heavily relies on floating-point computation and often involves very large/small numbers, posing a significant challenge of *numerical stability* of computation. We say a function is *numerical unstable* when a small change in the input can lead to large, and often unexpected differences, in the output [3, 11, 24]. For example, `division` is an unstable function—we can divide a value close to zero successfully; however, when the value becomes a bit smaller and is truncated to zero, the division reports an `INF` (when the divisor is zero) or `NaN` (when the dividend and divisor are both zeros). When such an error occurs, we say that *numerical instability* or *numerical bug* is triggered.

ML applications perform machine learning tasks typically using APIs from frameworks such as PyTorch or TensorFlow. The application code as well as the API implementations can contain unstable functions. When unstable functions are not properly guarded, there can exist a feasible input to trigger numerical instability in these unstable functions, which then causes the applications to crash, generate low accuracy and incorrect predictions, and/or stop learning but continue wasting computational resources during training [22]. The numerical bugs have been rated as *high priority* by professional ML developers [24].

Numerical bugs are challenging because (1) unstable functions commonly exist in ML libraries and applications [24] but ML developers may not know them; (2) even for ML experts, it is very hard to understand numerical properties of complex functions to add a proper assertion before invoking unstable functions; (3) in many cases, only special crafted inputs can trigger the bugs while the inputs can be complex tensors which traditional static analysis and testing tools cannot handle; and (4) numerical bugs do not always have a failure symptom of `NaN`/`INF`, and it can lead to an incorrect output that cannot be detected without specially crafted *oracles* (the oracle provides a ground truth output from a stable implementation).

In this paper, to address these challenges, we constructed a database of 61 unstable functions that can be referenced by ML developers. We analyzed the numerical properties and designed 6 types of oracles for the 61 unstable functions. We developed an automatic approach to encode safety/error conditions of invoking unstable functions via machine learning, using which we designed a novel fuzzing technique that successfully generates inputs to trigger numerical instability in ML applications.

Our database of unstable functions leveraged prior literature of DeepStability [24], where the authors identified unstable functions for PyTorch and TensorFlow libraries. We expanded the set of unstable functions and also provided more detailed information for each unstable function, such as oracles and potential input conditions that trigger the numerical bugs. There are GRIST [47] and RANUM [25] which can automatically detect numerical bugs in ML applications. Their work used manually encoded error conditions and thus only handled a few unstable functions. In addition, they only detected numerical bugs that led to `NaN` and `INF`. Our work aims to handle much more unstable functions via automatically encoded error conditions. We design test oracles that enable to find numerical bugs with failure conditions beyond `NaN` and `INF`. Other related work also includes PositDebug [5, 6]. It detects numerical bugs by using an alternative floating-point representation, *posit*, offering higher precision through shadow execution with high-precision values. DEBAR [50] uses abstract interpretation to analyze whether the value of a variable violates the expected range. They covered 8 unstable functions but faced the challenge of false positives.

Specifically, in this paper, we developed *Soft Assertion Fuzzer* to automatically detect numerical instabilities in ML applications. Our key novel idea is the *soft assertion*, aimed at using machine learning models to capture safety/error conditions of unstable functions. It addressed a critical challenge that developers cannot easily (in fact, sometimes impossibly) write symbolic assertions or

checks in the code to assure the numerical stability; such conditions require an in-depth understanding of numerical properties of library calls. In this paper, we first generate a soft assertion by testing an unstable function alone, such as a framework API. We then apply the soft assertions to guide fuzzing for any ML applications that use the unstable functions. The fuzzer will generate inputs at the ML application level to show how the numerical instability is triggered in ML applications, if any. Compared to existing fuzzers which typically used random input generation [18, 40] or code-coverage guided search [28], *Soft Assertion Fuzzer* mutate inputs based on the error conditions provided by soft assertions. We have shown, in our evaluation, that such assertion guided fuzzing is much more effective to trigger bugs than existing fuzzers.

Our techniques consist of two main components. The first component is the generation of soft assertions. For each unstable function we collected in the database, we automatically generated inputs and performed unit testing to collect successful and failure-inducing inputs for the unstable function. This execution profile data are then converted to the labeled data to train an ML classifier (soft assertion), where the input is the matrices input to the unstable function, and the label is a *soft assertion signal* that instructs how we should transform a passing input to trigger the numerical bugs. As an initial exploration, we used a set of predefined transformations, including *increase*, *decrease* and *no-change*, meaning that we should increase/decrease/not change the current value in order to trigger the numerical bug in the unstable function. We plan to expand to other labels in the future work.

Our second component is a fuzzer that uses soft assertions as guidance. Given an ML application, we first scan the code to identify the unstable functions. During fuzzing, when an execution reaches an unstable function, we activate its soft assertion to evaluate the current input values. If we obtain a *no-change*, we will execute the values to confirm that numerical instability is triggered; otherwise, we will use *auto-differentiation* [31] to propagate the soft assertion signal (the transformation advice) to the entry of the application code to inform how the current input should be mutated. While coverage-guided fuzzing simply mutates inputs that expand coverage, this soft assertion provides concrete guidance on which direction to mutate the input values. This iteration terminates until we trigger the numerical instability or until a timeout is reached.

We implemented *Soft Assertion Fuzzer* for Python programs that use PyTorch or TensorFlow frameworks. In evaluation, we constructed a benchmark consisting of 79 programs from GRIST [47] where we know the numerical bugs, and 15 real-world applications from GitHub with a 10 star rating, where we do not know the numerical bugs. We detected a total of 92 numerical bugs including all the 79 bugs in the GRIST benchmark and 13 unknown bugs in real-world code. Our tool significantly outperformed the 5 state-of-the-art Python fuzzers: (1) we found all the GRIST bugs, but more efficiently, (2) we found 13 real-world bugs, 1 of which already confirmed by the GitHub developers while the baselines hardly found any, (3) we found numerical bugs that lead to the incorrect output and incorrect ML predictions, while the baselines mostly handled the bugs that lead to `NaN/INF`. We presented a case study where a tumor detection model, TumorScope [35], trained on Brain MRI images incorrectly predicted "no tumor" after triggering a numerical instability.

In summary, our paper made the following contributions:

(1) We built a database, consisting of 61 numerically unstable functions—56 are collected from the prior literature [24] and 5 are discovered in our research. Compared to [24], our database included further details for each unstable function, such as oracles and potential input conditions that trigger the numerical bugs;
(2) We designed 6 types of oracles that enable us to find failures beyond `NaN` and `INF`;
(3) We proposed a novel idea of *soft assertion*, where we use ML models to automatically capture safety/error conditions of a numerical unstable function; our experience is that manually specifying symbolic assertions in these cases is almost impossible;

(4) We developed an approach to automatically infer soft assertions by testing numerically unstable functions against our oracles;
(5) We developed *Soft Assertion Fuzzer* that used the feedback of soft assertions to automatically generate inputs to trigger numerical bugs in ML applications. The user can directly use our tool to scan their code for numerical instability supported by soft assertions we generated so far. The user can also expand to more unstable functions by generating soft assertions using our framework and our approaches, and then plug them into our fuzzer;
(6) We performed a comprehensive evaluation on 94 ML applications and 5 SOTA baseline fuzzers, and demonstrated that our tool can find numerical instabilities in real-world ML code where existing tools cannot. We also show that besides `NaN/INF`, numerical bugs can lead to incorrect model predictions (we have not seen such examples in prior literature).

2 A Motivating Example

```
if __name__ == '__main__':
  //Input tensor: x, y
  dist = pairwise_distances(x.abs(), y.abs())
  trans, cost = compute_optimal_transport(dist)
  ...
def pairwise_distances(x, y, method):
  ...
  if method == 'l2':
    dist = torch.pow(x - y, 2).sum(2)
  elif method == 'cosine':
    dist =  1 -  nn.CosineSimilarity(x, y)
  return dist

def compute_optimal_transport(M, l=0.2):
  ...
  P = torch.exp(-l*M)
  ...
  transport_plan = P/P.sum(-1)
  cost = torch.sum(P*M)
  return transport_plan, cost
```

Fig. 1. Unstable `CosineSimilarity` leads to incorrect results in PyTorch

In Figure 1, we show a numerical bug in a snippet of machine learning code that triggers the unstable function, `CosineSimilarity`, implemented in PyTorch 2.2.1. The code takes two tensor inputs that represent a source and a target; it aims to find an efficient strategy to allocate resources from the source to the target. The numerical bug has led to an incorrect output of this program and incorrectly computed the cost of resources. For example, given the input:

$$x = \begin{bmatrix} 2606.66824394 & 2477.72226966 & 3251.84008903 \end{bmatrix}$$

$$y = \begin{bmatrix} 1.4682216 \times 10^{-9} & 1.825367 \times 10^{-8} & 8.3524 \times 10^{-9} \end{bmatrix}$$

the correct output `cost` should be 0.08963637, but the actual output `cost` is 0.1601. We used an oracle to determine the correct output. This oracle compared the PyTorch implementation with a more stable implementation of cosine similarity in NumPy. See Section 3.2.2 for details.

Analysis of the numerical bug: The two inputs, `x` and `y`, represent a set of characteristics for the source and the target respectively. The `pairwise_distances` function at line 3 first evaluates the discrepancies between the two inputs. At line 4, this distance input is passed to the `compute_optimal_transport` function. This function (details at line 14) takes the input of discrepancies and report `P` at line 16; `P` represents how to redistribute resources from the source to the target in a manner that can minimize the total cost. Finally, `transport_plan` is generated after normalizing `P` at line 18, and `cost` is calculated at line 19.

The root cause of this error is that the input triggered the instability of `CosineSimilarity`. The cosine similarity between two non-zero vectors can be calculated using the formula for the Euclidean dot product:

$$cosine\_similarity(x, y) = \frac{x.y}{||x|| * ||y||}$$

When the *norm* ($|| \cdot ||$) of any input is less than 10^{-8}, the PyTorch implementation will assign the norm value to 10^{-8}. In this example, the norm of $||y||$ should be 9.2263×10^{-9}, but $||y||$ is adjusted to 10^{-8}. As a result, the cosine similarity results in 0.8399 instead of 0.91036362. This deviated value of `dist` at line 11 is then propagated in the code and lead to the incorrect output of `cost` at line 20. In the past, other researchers [24] have also reported the instability of cosine similarity, where the bug produced an output larger than 1, outside its valid range of [-1,1].

Challenges of avoiding and detecting such bugs: This type of numerical bug is hard to avoid or detect; unlike a divide-by-zero error where a developer may know to add a check or assertion before the division, stating that *the divider should not be a zero*, here, it is not easy to write a check or assertion before or after `CosineSimilarity` to detect the error. This is because some ML functions are mathematically complex, and it is challenging to understand when the function could become unstable [24]. Existing work on mining symbolic preconditions, e.g., Daikon [7], cannot work here either (see Section 6)—- in ML applications, the stability conditions involve high-dimensional matrices and vectors, e.g., it is hard to express "the determinant of a high-dimensional matrix (of unknown size) is zero" in a symbolic condition.

Our novel idea to address the challenges: Although it is hard to express the condition of numerical stability as symbolic constraints, we trained an ML model, namely *soft assertion*, to encode such conditions and use it as an assertion to detect numerical instability and give feedback to guide fuzzers. Soft assertions are inserted at the call to a numerical unstable function, which can be either an API to the library or a user-defined function. Soft assertion is an automatically generated ML model that encodes the error-triggering conditions for an unstable function. The training dataset consists of failure-inducing and successful inputs automatically obtained via performing unit tests of an unstable function. For each unstable function like `CosineSimilarity`, the training is only performed once to generate its soft assertion, and once generated, the soft assertions can then be repeatedly used in any ML applications that use the function. Like typical program assertions, soft assertions may be used for runtime monitoring and/or serving oracles for testing; in this paper, we focus on exploring its applications for detecting numerical instability and effectively guiding fuzzers to trigger numerical bugs in ML applications.

3 Soft Assertion

3.1 What is a Soft Assertion?

Soft assertion is similar to a typical symbolic assertion in that we insert it at a check point of a program to detect errors. The term 'soft' reflects the fact that these assertions are not precise symbolic conditions; instead, it encodes (approximate) error conditions in a machine learning model when symbolic conditions cannot be easily specified. We view "soft assertion" as one kind

of "assertion" because it takes program values at the assertion point and reports whether the error condition is violated. When the assertion is not triggered, it provides advice on whether the current values should be *increased* or *decreased* in order to trigger the assertion. Such functionalities are similar to symbolic assertions, which can also be used to trigger failure conditions and guide testing.

In this paper, we apply soft assertions to find numerical bugs, and therefore, soft assertions will be inserted before unstable APIs (the unstable functions implemented in the library) or calls to the user-defined unstable functions in an ML application. We have collected a list of unstable functions through our research and also from prior literature [24]; the users can also use our approach to generate soft assertions for their own defined unstable functions.

Definition: A *Soft Assertion (SA)* at a program point p is a machine learning model that takes the program values, denoted as X, at p, and reports how to transform X to trigger the error condition at p. Mathematically, we denote

$$SA(X) = \begin{cases} increase \\ decrease \\ no\ change \end{cases} \tag{1}$$

where X consists of program values at p. The output $SA(X)$ reports three types of actions: *increase* (increasing the values in X), *decrease* (decreasing the values in X) and *no change* (the error triggered and the failure-inducing values found). We call the output of the soft assertion, $SA(X)$, *Soft Assertion (SA) Signals*. Our current design of formula (1) tailors the applications of guiding test input generations in fuzzers. In the future, we plan to expand soft assertions to more classes and explore applications beyond fuzzing.

We designed the concept of soft assertion by referencing the roles of a traditional symbolic assertion. An assertion can test if the error condition is satisfied, which corresponds to the *no change* signal in formula (1). Traditional symbolic assertions can also be incorporated by test input generators, such as symbolic execution, as constraints to generate test inputs. Considering that writing symbolic assertions for high dimensional tensor values and complex numerical functions are extremely challenging, and symbolic execution cannot be easily applied, in this paper, we apply fuzzing for test input generation and explore alternative signals (instead of concrete symbolic conditions) to guide test input generation. Specifically, we used a simple classification formulation and designed *increase* and *decrease* two SA signals, as show in formula (1), to provide the directions of input mutation in fuzzing, as we are fully aware that it is very difficult to require an ML model to generate complex symbolic conditions [44, 45]. Later, we show in Section 5, even with our simple formulation, SA signals can be effectively used to mutate program inputs to trigger the bugs.

3.2 How to Generate a Soft Assertion

In this paper, we developed an approach to automatically generate soft assertions. It only requires the unstable function as input, and the generation is independent of the ML applications that call the unstable functions. The soft assertion of an unstable function only needs to be trained once, and then it can be inserted in different ML applications where the unstable function is called.

In Figure 2, we show our approach, which consists of four steps. First, we collected a list of frequently used, known unstable functions (Section 3.2.1). To build a soft assertion, we developed the oracles to determine whether the numerical bugs were triggered at the end of the numerical function (Section 3.2.2). Using the oracles, we ran unit tests on individual unstable functions to record the failure-inducing and successful values at the entry of each unstable function. In the next step, we designed a test input generation technique to efficiently generate the balanced training dataset consisting of successful and failure-inducing inputs (Section 3.2.3). Finally, we trained a

ML model for each unstable function that can predict the actions of $SA(X)$ in formula 1. In the following sections, we will describe each step in detail.

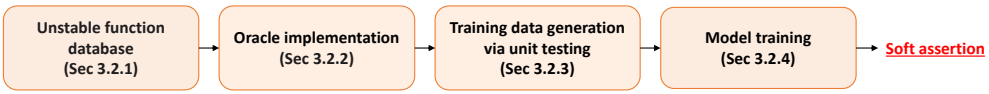


Fig. 2. An Overview of Soft Assertion Generation

3.2.1 Building a Database of Unstable Functions. Machine learning applications frequently invoke APIs from the libraries and frameworks. Numerical errors can be largely managed if we can identify frequently used unstable APIs and ensure that when they are called in applications, their instability (error conditions) cannot be triggered.

We studied the prior literature [24] and collected a total of 56 unstable functions from the PyTorch and TensorFlow libraries. During the study, we found additional 5 unstable functions, and our database thus includes a total of 61 unstable functions. These functions are critical to ensure numerical stability and performance in modern ML applications. The functions consist of activation functions, derivative computation and *hyperbolic* functions (see our replication package for the entire list of functions). For each unstable function, we attempted to manually identify input conditions that trigger numerical bugs and succeeded for 11 functions. We have tried the binary search on the input range and used extremely large and small boundary values as well as diverse input patterns of a matrix, e.g., sparse matrices and ill-conditioned matrices. It is consistent with what we mentioned above that it is hard to create a symbolic assertion or insert a checker in the application code to avoid the numerical errors of the unstable functions being triggered. While the prior literature listed many of those unstable functions [24] and explained why they are unstable for some functions, our work analyzed all 61 unstable functions in our database, and identified test oracles for these functions.

3.2.2 Designing and Implementing Oracles. To generate soft assertions, we need to identify the oracles that can distinguish the correct and incorrect output of unstable functions. To design the oracles for numerical bugs, we first conducted a thorough analysis of the mathematical properties and potential sources of instability of the 56 unstable functions in our database, using the prior literature [24, 47, 50] as a reference. This analysis allowed us to identify specific types of numerical errors and failure symptoms these functions might produce such as undefined results, out-of-range results, precision loss/incorrect results, algorithm instability, and inconsistencies across implementations that use different libraries. For each error category, we formulated a type of oracle designed to detect that specific class of error. Our analysis resulted in 6 types of oracles, summarized in Table 1. In the following, we provide the detailed explanations:

(1) NaN/INF detection: Some unstable functions fail with a `NaN` or `INF`, indicating undefined or infinite results. We invoke `isNaN and isInf` (implemented in the ML libraries) to test the return value of the unstable function to determine if the unit test result leads to a NAN/INF. For example, a division of a very small number can lead to a NAN/INF. Here, the division is an unstable function, and the NAN/INF check is used as an oracle. We also identified `SoftMax` and `L2Norm` as this category based on our analysis of these functions.

(2) Output-of-range check: For some unstable functions, we can identify a valid range. For example, `sin(x)` and `cos(x)` have a valid range of [-1,1], any deviation from the values will be flagged as undesirable output. Sometimes, even if the return value is within the valid range, numerical instability can still occur as shown in the example of Figure 1, so we will consider all types of oracles that apply for an unstable function.

Table 1. Constructing Oracles to Test Unstable Functions

Oracle	Example Unstable Function	Math Properties of Unstable function
NaN/INF detection	Softmax, L2 Norm, Division	The function has a division operator; the divisor can be small or 0
Out-of-range check	Sin(x), Cos(x), Inverse tangent, Hyperbolic tangent	The function has known bounded outputs.
Math formula rewriting	x-(max + log(y)) x-y × log(x)	The function has a subtraction operator, and the operands have very similar values; or it contains a log operation that can take very large/small input.
Stable algorithm implementation	Matrix Inverse, Matrix determinant, Eigen value decomposition	The function uses complex algorithms that are highly sensitive to ill-conditioned matrices or matrices that are nearly singular.
Consistency check across frameworks	Cosine similarity, Linear solvers, Eigen value calculation, SVD, Random number generator	Some functions are known for their sensitivity to numerical stability, so we need to check across different library implementations
Increase width	Remainder, Fast Fourier transform, polynomial roots, PCA	The function uses operations that can lead to precision loss when the values of operators have different magnitudes (small/large input).

(3) Math formula rewriting: Some math formulas are unstable, but we can create more stable implementations through rewriting [24]. For example, we can rewrite $x/(sqrt(x) * sqrt(x))$ to $x/sqrt(x * x)$ to avoid precision loss, and rewrite $x - (max + log(y))$ to $x - max - log(y)$ to avoid the loss of significant digits. We implemented a stable version of the function using math formula rewriting as an oracle. We compare the output from the current unstable equation with the output from the oracle.

(4) Stable algorithm implementation: Some algorithms may produce mathematically accurate results but lead to numerical issues for some input. For example, the algorithm used to calculate the direct inverse of matrix exhibit instability. If the input matrix is ill-conditioned such as the matrix is close to the singular matrix, the result can be incorrect. Here, the `inverse` operation is the unstable function, and we use a more stable implementation as an oracle [24] for the inverse calculation, namely *Cholesky inverse*.

(5) Consistency check across frameworks: Different library frameworks sometimes have different implementations for the same math functions, resulting in different numerical stability properties for the functions. For example, we found that some inputs to the `cosine similarity` function can lead to very different outputs across NumPy, SciPy, and PyTorch. Compared to our manually calculated results, we found that NumPy implementations are the most accurate and thus we used it as oracles. In a pilot study, we generated two 1024 vectors of randomly generated values ranging from 10^{10} and 10^{-10}, our manually calculated values and NumPy both report 0.753396; however, PyTorch reports 0.137953. When implementing this oracle, we are aware that we cannot

always exactly match the two floating point numbers even if they are both correct; therefore, we consider a match when the return values of two implementations have a difference within a configurable threshold (we set 0.000001 in our experiments following the literature [9]).
(6) Increase the width of floating-point representation: For the cases where none of the above oracles are applicable, we consider to increase the width of the floating point numbers. For instance, a remainder calculation with 32-bit floats might not accurately represent certain decimal numbers. As an example, `1933053808.0%53` leads to 35 instead of the correct value 19 when calculated in the 32-bit precision. Thus, in the oracle, we implement the `remainder` operation using the 64-bit precision to represent its operands.

Table 2. Numerical Stability Analysis of Newly Identified Functions

Function	Safe Condition	Source of Instability	Oracle
Linear Solver ($Ax = b$)	Unknown	(1) Large condition number (2) Near-singular matrix ($det(A) \approx 0$)	Increased width
SVD (Singular value decomposition)	Unknown	(1) Near-zero singular values (2) Ill-conditioned input matrix (3) Loss of orthogonality	Consistency check across frameworks
Eig (Eigenvalue Computation)	Unknown	(1) Ill-conditioned eigenvectors (Repeated eigenvalues, large eigenvalue ratio)	Increased width
SELU/ELU	-103.9720 $\leq x_i \leq$ 3.40e38	(1) Certain negative inputs in tensor (vanishing gradient) (2) Large positive inputs in tensor (overflow)	NaN/INF detection
GlorotUniform (Weight Initialization)	Unknown	(1) Improper scaling (2) Poor weight distribution	Stable algorithm implementation

Table 2 provides an overview of five functions identified in our research that were not reported in prior works. We list the rest of unstable functions in our replication package, including the oracle type(s) for each unstable function. We implemented oracles for 25 unstable functions. In the future, if the user wants to extend our framework to support a new unstable function, they can determine which one(s) of the 6 oracles matches the potential numerical errors the unstable functions will produce. It will be useful to analyze the mathematical properties of the function, for example, the domain and range of the function, or the potential sources of instability such as subtraction of similar numbers, and division by small values). See the third column in Table 1 for the mathematical properties we considered when determining oracles.

3.2.3 Performing Unit Testing on Unstable Functions to Create Training Data. Our goal is to train an ML model that approximates formula (1), defined in Section 3.1. Specifically, the model input X is the test input of the unstable function, and the model output is the transformation we should use to mutate X to trigger the numerical instability in the unstable function. In Table 3, we used examples to explain how we produce the training data using unit testing of unstable functions.

Suppose we randomly generate a test input 10 (note that in real applications, it will be a matrix), namely the *base* input, as shown in the first row of the table. When running this input, the numerical

Table 3. Our approach of generating training data

Base	Mutation	Current	Output Change	Training Data
10	increase	30	fail → success	(30, decrease)
				(10, no change)
-1	increase	5	success → fail	(-1, increase)
				(5, no change)

instability of the unstable function is triggered. We then apply a mutation to this input(without the loss of generality, let's say it becomes 30). This input leads to a success in testing. Through these two tests, we obtain two training data points: (1) (30, decrease), meaning that given a successful input 30, if we decrease it (e.g., to 10), the unstable function likely fails; and (2) (10, no change), meaning that given 10, the unstable function will fail and no change is needed. Similarly, in the second example, shown in the next row, our base input is -1. Suppose it leads to a successful execution. After mutation, when we increase its value to 5, the test fails. We obtain the training data points (-1, increase) and (5, no change).

We aim to generate a balanced training data set, i.e., obtaining the similar numbers of labels of "increase", "decrease" and "no change", and we also hope the input set can cover a diverse set of values. We developed two key components (1) the approach of mutating inputs, and (2) the approach of generating initial base inputs used for mutations:

(1) Mutation approaches: The goal of the input mutation is to obtain a new input whose test outcome would differ from the base input, i.e., one fails and the other passes during unit testing of an unstable function. The transformation between the two inputs can then be used to generate labels as indicated in Table 3.

Triggering numerical stability is very challenging, and our mutation supports a set of configurable methods to do the step-wise increases/decreases, starting from the base input. Specifically, we implemented the *exponential* ($step = e^{rate}$), *random* ($step = r * rate$), and *sinusoidal* ($step = sin(rate)$) three methods for choosing different speeds of changes during mutation. *rate* here is a configurable parameter. To further increase our chance of obtaining diverse data, we mutate from failure-inducing inputs to success inputs (shown in the first row in Table 3), and from successful inputs to failure-inducing inputs (the second row in the table).

Here we only used the basic transformations of *increasing*, *decreasing* and *no change* as labels. In the future, we plan to also consider quantized values to provide more fine-grained transformation labels, e.g., *increasing 100* is a label, and *increasing 200* is another label. This approach potentially leads to more accurate guidance for fuzzers.

(2) Base input generation: To generate initial base inputs, we sample n values (n is configurable) across different regions of the input space. For example, we can sample 10 inputs less than 0, 10 inputs between (0,100), and 10 inputs larger than 100. If we have the domain knowledge of an unstable function, we can sample regions more efficiently, e.g., directly using some failure-inducing inputs we know. In our experiments, we set n to 100.

Next, we apply the above mutation techniques to generate new values and test them on the unstable function. As soon as we observe the change of the test outcome for input x and its mutated input x', we record training data points like the ones stated in Table 3. For example, given an initial value 10 (test successes) and its mutated sequence, 20 (test successes), 30 (test successes), 40 (test fails), we collect the training data points (20, increase), (30, increase) and (40, no change) respectively; similarly, given an initial value 100 (test fails), and its mutated sequence, 130 (test fails) and 160 (test successes), we collect the training data point (160, decrease), (100, no change)

and (130, no change) respectively. The overall complexity of this process is $O(n \cdot m)$, where n is the number of base inputs, and m is the number of mutations per base input. Our empirical results show that this method can generate failure-inducing inputs for the majority of functions under study. We also added known failure-inducing inputs from the literature [24] to increase the chance of triggering failures.

Our inputs support any size of vectors and matrices. In the experiments, we used randomly generated base inputs as elements of matrices, and we also perturbed the real-world data such as MNIST to obtain inputs. For the real-world dataset, we make sure that the mutated inputs still follow the constraints of image pixels (0,255).

Sometimes, when we test randomly generated matrices or a perturbed real-world dataset directly against our unstable functions, the tests always fail. In this case, we perform a data pre-processing to scale the data to make sure that some inputs fail, and some are successful. In this way, we can obtain a balanced dataset to train the model. For example, a lot of pixels of the MNIST image have the value 0 (black colored). We added a small ϵ to ensure when we test an unstable function, it is not always crashing.

3.2.4 Training a Machine Learning Model to Generate a Soft Assertion. Once we obtain the training data, we train a machine learning model that takes matrix/vectors as inputs and reports the classification specified in formula (1). We explored a variety of machine learning model architectures, including DNN, RNN, LSTM, transformers and K-Nearest Neighbors (KNN) and Random Forest. We found that Random Forest can handle any range of random values without encountering numerical stability itself [43]. It also achieved highest accuracies for majority soft assertions we generated, without the need of pre-processing of data. See Section 5.1.1 for the implementation details of model training.

4 Fuzzing with Soft Assertions

4.1 The Fuzzing Framework

Figure 3 shows an overview of our *Soft Assertion Fuzzing* framework. The fuzzer takes an ML application as input. It scans the code and compares our *Unstable Function Database* to locate the unstable functions in the ML application. The fuzzer starts with random inputs and during execution of the inputs, it invokes the *Soft Assertion Checker* to take the values at the entry of the unstable function and make a prediction using soft assertion. If the error triggering condition is met, the fuzzer continues the execution to demonstrate the error; otherwise, the *Soft Assertion Checker* returns a *Soft Assertion (SA) Signal* to teach the fuzzer how to transform the current value to trigger the bug, e.g., increasing or decreasing the values. This proposed transformation is propagated to the input via the *Auto-diff* component to guide the mutation to generate the input for next iteration.

4.2 The Fuzzing Algorithm

In Algorithm 1, we present our fuzzing technique that uses soft assertions. The input of our tool is (1) a machine learning application, p, (2) the unstable function database D (Section 3.2.1), and (3) their soft assertions, A, we trained for the unstable functions (Sections 3.2.2—-3.2.4). The output is the failure-inducing inputs, I, that can trigger numerical bugs for the unstable functions in p.

At line 2, we first scan the code to identify unstable functions in program p. For each identified unstable function f, we search for inputs that can lead to its failure starting at line 3. At line 4, we first randomly generate an initial input for x, and at line 6, the algorithm iterates throughout a loop, where we mutate the input in successive iterations to search for a failure-inducing input. $T_{timeout}$ is a configurable timer we used to control fuzzing time. At line 5, we initialize H, which will store the history of the inputs tried during fuzzing. In particular, we store the inputs at each

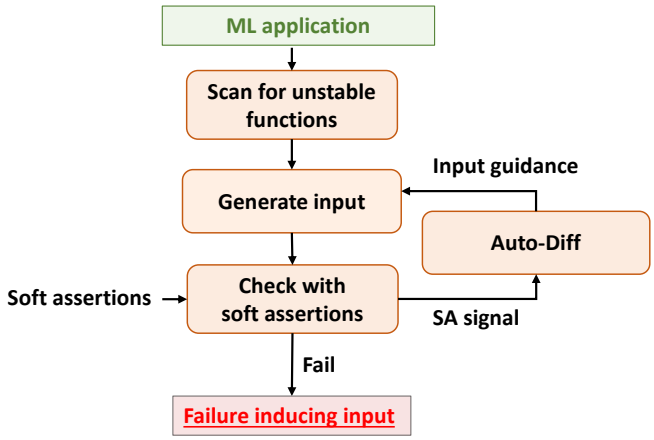


Fig. 3. An Overview of *Soft Assertion Fuzzer*

Algorithm 1 Soft Assertion Fuzzing Algorithm

Input: Program p, Unstable function database D, Soft Assertions A
Output: Failure-inducing inputs I

```
1:  I ← ∅
2:  F ← ScanForUnstableFunctions(p, D)
3:  for f ∈ F do
4:      x ← GenerateInitialInput()
5:      H ← ∅
6:      while t < T_timeout do
7:          v ← Execute(x, p, f, entry)
8:          signal ← SoftAssertion(A, v, f)
9:          if signal == NO CHANGE then
10:             if Execute(x, p, f, exit) == SAFE then x ← GenerateInitialInput()
11:             else
12:                 add x to I; print t; break
13:             end if
14:         else
15:             Δx ← AutoDiff(x, f, signal)
16:             x′ ← ConstraintSolving(x, Δx, H)
17:             add ⟨x, Δx⟩ to H; x ← x′
18:         end if
19:     end while
20: end for
21: return I
```

iteration together with the update instructions we propagated from soft assertion signals as well as the test outcomes. This history provides us constraints to search for inputs of next iterations. For example, from the history (10, increase, success) and (100, decrease, success), we can obtain the constraints to generate failure inducing inputs $10 < x < 100$, and then we can use constraint solver to generate the input.

At line 7, we run the current input x on p till it reaches the entry of f, where we record the value v. At line 8, we take this value and query the soft assertion of f. If the returned soft assertion signal suggests a failure with the label *no change* (see line 9), we perform a validation by continuing executing x on p till the end of f. At line 10, we compare the output at the exit of f with the oracle. If the oracle reports that the current input x is safe, indicating a misprediction by the soft assertion, the process is reset with a new initialized input. However, if the oracle confirms that the execution fails with x, we add x to the failure-inducing input set I, and print the time used, t, at line 12. This concludes the search for the current function f, and we then move on to the next unstable function in F.

If at line 9, the soft assertion does not return `NO CHANGE`, we will need to mutate the current input x towards the areas more likely to uncover failure-inducing inputs. The soft assertion signals are designed for providing such feedback. At line 15, we propagated the soft assertion signal generated at f back to the entry of the program using an auto differentiation component (details in Section 4.3). The adjustment d computed from line 15 will be used to update the current input x to generate a new input x'. In the process of updating the current input x, shown at line 16, we incorporate the history of previous inputs, denoted by H, which is initially set up at line 5 (details in Section 4.3). Subsequently, at line 17, both x and d are added to the input History H, and the new input x' is set to x, and will continue in next iteration and be executed at line 7.

4.3 Using Soft Assertion Signals to Guide Input Generation

Soft assertion signals (generated at line 8 in Algorithm 1) instruct, at the assert location, how to adjust the current values to trigger the instability in the unstable function. We formulate the details of our approach in math as follows. Given an input x and a section of code, annotated as g, before reaching the unstable function f, the value y at the entry of f is $g(x)$. We can use *auto-differentiation* [2] to generate the output of $g'(x)$, where g' is the derivative function of g. $g'(x) = \Delta y/\Delta x$ captures the rate of the change of y around a particular value of x. When the soft assertion outputs a signal on how to transform y to trigger instability, i.e., Δy, we can then compute the corresponding update of input Δx using $\Delta y/g'(x)$.

Considering training a machine learning model to predict the concrete value of Δy can be hard, in this paper, we explore the feasibility and effectiveness of using the directions of value changes, such as "increase" and "decrease"; that is, in the *AutoDiff* function in Algorithm 1 at line 15, we implemented the following formula:

$$\Delta \mathbf{x} = \frac{SA(X) \times \text{Rate}}{\nabla_{\mathbf{x}} g(\mathbf{x})} \tag{2}$$

Here, $\Delta \mathbf{x}$ represents the adjustment to program input $\mathbf{x}$, $SA(X)$ is the direction suggested by the soft assertion defined in formula (1), $\nabla_{\mathbf{x}} g(\mathbf{x})$ is the gradient from automatic differentiation (AD), and *Rate* adjusts the mutation's intensity, which is configurable in our tool.

Leveraging Input History for Mutation: In Algorithm 1 at line 16, we used the history of generated inputs to help determine the values of a new input. Specifically, we construct a set of constraints based on the pairs of $\langle x, d \rangle$ stored in H at line 18. For example, for $\langle 30, decrease \rangle$, we add $x < 30$; for $\langle -1, increase \rangle$, we add $x > -1$. We combine these constraints with formula (2). to generate a new input.

5 Evaluation

To evaluate our *Soft Assertion Fuzzer*, we studied the following research questions:

- **RQ1:** Can we effectively generate soft assertions?
- **RQ2:** Can our *Soft Assertion Fuzzer* effectively detect numerical bugs?

- **RQ3:** In what aspects, does *Soft Assertion Fuzzer* perform better than the state-of-the-art techniques?
- **RQ4:** How do different configurations affect the effectiveness of *soft assertion fuzzer* (ablation studies)?

5.1 Experimental Setup

5.1.1 Implementation. We implemented our *Soft Assertion Fuzzer* in Python 3.10.12, the most stable and recent version. It consists of a lightweight static analysis tool that scans the code and compares it with the unstable function database to locate unstable functions in the code. It invokes soft assertions during fuzzing in these unstable functions. We currently can handle programs that use PyTorch or Tensorflow libraries. We used the *PyTorch Autograd* package, which can handle ML applications that use *PyTorch* libraries, and *Tensorflow*'s autodiff methods, called *Gradients*, which can handle Tensorflow libraries. We conducted our experiments on Python benchmarks, but we believe that with small modifications, *Soft Assertion Fuzzer* can also work for Java or R programs.

Out of 61 unstable functions in the database, we implemented soft assertions for 25 representative functions. We included all the unstable functions GRIST [47]handled, including `log`, `sqrt` and `exp`, so that we can compare the GRIST benchmark. We ensure that the 25 unstable functions cover a variety of types, from mathematical operations (e.g, `Reciprocal` and `Division`), activation functions (e.g, `Softmax`, `Sigmoid` and `ReLU`), to hyperbolic functions (e.g, `Tanh` and `Sinh`), so that our tool can benefit a variety of ML applications. To demonstrate the advantage of soft assertions, we especially sampled the functions, e.g., `Matmul` and `Mean`, where we do not have knowledge on their safety input conditions, and it is very hard to add a traditional symbolic assertion to protect those functions. To evaluate whether the ML models are capable of encoding error conditions of complex math functions, we considered a variety of math functions frequently used in the neural networks, including the inverse trigonometric functions (e.g, `Acos`), exponential operations (e.g, `Power` and `Exp`), `Cosine Similarity` and `Cross Entropy` as well as `Linear` and `Conv2d`.

To generate soft assertions, we implemented a set of neural network architectures and ML models such as DNN, RNN, LSTM, and transformers using PyTorch, as well as KNN and Random Forest using Scikit-learn. We did pilot studies with these models on a subset of unstable functions under study, and found that Random Forest performed the best. Specifically, we used RandomForestClassifier from Scikit-learn with 100 decision trees, random selection of data subsets is 42. For each unstable function, we trained a model, using automatically generated datasets of the size 40 k (30% of these datasets are used for testing). We explored several settings of input data, including the matrices of 3×3, the matrices of 14×14 (half of the size of an MNIST image) as well as MNIST. We also explored, in our pilot study, to use the CIFAR and Fashion-MNIST datasets for unstable functions, but we found that the model accuracies are not as good compared to smaller matrices. We trained each model for each dataset three times and logged the average F1 score and training time. Note that model training is a one-time process, and once done, it can be repeatedly used in different ML applications.

5.1.2 Baselines. We used five SOTA of Python fuzzers as baselines, including *GRIST* [47] and *RANUM* [25] targeting numerical bugs in ML applications, *PyFuzz* [18], a Python fuzzer released in 2022, *Atheris*[28], a well-maintained fuzzer developed by Google for coverage-guided fuzzing for Python, and *Hypothesis* [40], a widely used framework for property-based testing.

We used the default settings and hyper-parameters to run the baselines. We ran these fuzzers 5 times for each program and get an average time as the results. We set the timer for 30 minutes for each program for all the fuzzers including ours.

Table 4. Generating soft assertions: F1 scores and training time in minutes

No.	Unstable Func	3×3 matrix	Train time	14×14 matrix	Train time	MNIST	Train time
1	Softmax	**0.8145**	0.286	0.8106	0.998	0.7107	0.796
2	log	**0.6574**	0.282	0.6172	1.149	0.5018	0.320
3	sigmoid	**0.6210**	0.360	0.6160	1.188	0.5039	0.948
4	exp	0.6663	0.115	**0.8101**	1.001	0.7038	0.804
5	logSftmx	0.5518	0.481	0.6289	1.274	**0.7136**	0.776
6	sqrt	0.6134	0.319	0.6343	1.645	**0.8011**	3.567
7	tanh	0.5411	0.349	0.5334	1.688	**0.7241**	0.756
8	ReLU	0.6136	0.352	**0.6171**	1.709	0.5424	1.105
9	ELU	**0.6469**	0.348	0.6421	1.900	0.5441	0.761
10	SoftPlus	0.6228	0.370	**0.6267**	1.141	0.6034	2.482
11	rSqrt	**0.6204**	0.392	0.5977	4.869	0.5295	1.099
12	Div	0.6215	0.359	**0.6255**	1.803	0.5426	0.470
13	linear	**0.8141**	0.241	0.7870	1.422	0.7141	1.378
14	matmul	0.6657	0.298	0.6562	1.759	**0.7134**	1.379
15	mean	0.5272	0.255	**0.6661**	1.353	0.5524	1.262
16	reciprocal	0.6376	0.380	**0.6427**	2.050	0.5630	0.375
17	CosSim	0.6503	0.342	**0.6632**	1.366	0.5062	0.394
18	acos	0.6681	0.241	0.6673	1.278	**0.6810**	0.416
19	cosh	0.8092	0.277	**0.8133**	0.980	0.7644	0.675
20	sinh	0.8119	0.279	**0.8134**	0.993	0.5094	1.248
21	square	**0.6702**	0.243	0.6631	1.397	0.5028	1.215
22	pow	**0.6667**	0.240	0.6590	1.318	0.5549	2.645
23	sum	0.6601	0.254	**0.6689**	1.290	0.5601	1.761
24	CE	0.6380	0.370	**0.6447**	1.981	0.5472	1.329
25	Conv2d	0.6347	0.372	**0.6415**	1.945	0.5396	1.340
	Average	**0.6577**	0.312	**0.6698**	1.580	**0.6051**	1.172

5.1.3 Constructing a Benchmark of ML Applications. To evaluate our techniques, we constructed a benchmark consisting of 94 ML programs, among which, 79 programs are from the GRIST benchmark [47], where we know existing numerical bugs (useful to evaluate the false negatives of our tool). GRIST has also been used by evaluating other recent tools, e.g., RANUM [25].

Additionally, we selected 15 programs from various GitHub projects where we don't know the bugs, which helped evaluate the usefulness of our tool for real-world code. To find important and recent projects in GitHub, we used GitHub API to search for Python code. Our query targets repositories using the NumPy, TensorFlow or PyTorch framework, focusing on the implementation that used unstable functions by providing these keywords such as `sqrt`, `exp` and `Softmax`. From the search results, we selected 30 repositories that have at least 10 stars to make sure we focus on potentially impactful projects. We manually reviewed these repositories and kept the ones that have been updated within the last three years. We also excluded the ones that potentially use incompatible Python versions and obsolete frameworks. Sometimes, the given keyword is mentioned in the comment section but not implemented in the code; we exclude those cases as well. After filtering, we obtain a total of 15 programs as the subjects for our evaluation.

5.1.4 Running Experiments. All of our soft assertions are trained on a single NVIDIA (A100-SXM4-80GB) GPU, supported by an x86_64 architecture CPU with 251.35 GB of RAM. We conducted our fuzzing experiments on the Intel(R) Core(TM) i5-8365U CPU @ 1.60GHz with 16GB RAM, Ubuntu 22.04.6 LTS. We used the Anaconda environments to switch between versions of PyTorch and TensorFlow for the benchmark programs that use different library versions.

5.2 RQ1: Generating Soft Assertions by Testing Unstable Functions

In Table 4, we report our results on soft assertion generation for 25 unstable functions. We used three sizes of input, including 3×3 *matrix* and 14×14 matrix as well as the perturbed MNIST (28×28 matrix), following Section 3.2.3. Across 25 unstable functions listed in the second column, we achieved the average F1 scores of 0.6577, 0.6698 and 0.6051 respectively for the three settings. Some soft assertions are accurate and achieved 0.8145 F1 score (Column 3×3 *matrix* Row *Softmax*) and 0.8101 F1 score (Column 14×14 *matrix* Row *exp*). We did not observe the advantages of one setting over the others. The accuracy seems to be dependent on individual unstable functions. Although the F1 scores of these soft assertions are not always very high, we found that they already are very useful for helping fuzzers trigger numerical bugs (see Sections 5.3 and 5.4). We also reported the training cost for each unstable function. The last row in Table 4 shows that the models were trained fast and can finish 0.312–1.580 minutes on average for the three types of input data.

5.3 RQ2: Detecting Numerical Bugs in ML Applications

Table 5. Detect numerical bugs for real-world programs: (1) compare with baselines, (2) ablation studies

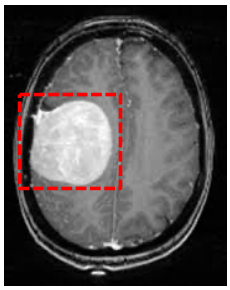
Github Repo	Failure Symptoms	Comparing with Baselines (Time in seconds)					SA Fuzzer	Ablation Study					
		PyFuzz	Hypo	Atheris	GRIST	RANUM		No H	MNIST	DNN	3×3		
											40K	15K	5K
[17]	NaN in `rqst`: negative pixel values after normalization	×	×	×	×	×	**0.0243**	×	0.3320	0.4200	0.0645	0.1289	0.1935
[46]	×	×	×	×	×	×	×	×	×	×	×	×	×
[1]	×	×	×	×	×	×	×	×	×	×	×	×	×
[12]	INF in `exp`	×	0.0438	×	140.8300	1.4400	0.0660	0.0070	**0.0070**	0.0123	0.0070	0.0133	0.0210
[37]	×	×	×	×	×	×	×	×	×	×	×	×	×
[8]	NaN in `power` caused by overflow	×	×	×	×	×	0.0150	0.0090	0.0084	0.0139	**0.0080**	0.0159	0.0253
[38]	-INF in `log` for very small probabilities (close to zero)	×	×	×	18.6500	0.6180	0.0070	**0.0030**	0.0170	0.0192	0.0170	0.0323	0.0510
[23]	NaN in `matmul` caused by overflow	×	×	×	×	×	**0.1840**	1.1300	0.9440	1.4500	0.9400	1.7920	1.8320
[27]	NaN in `L2 norm` caused by overflow	×	×	×	×	×	**0.0100**	0.9000	1.7950	2.0500	1.7000	3.4105	3.4565
[29]	INF in `matmul`	×	×	×	×	×	**0.2480**	1.3000	1.2700	1.1400	1.2700	2.4110	2.8580
[10]	INF in `exp`	×	×	×	×	×	**17.0500**	27.7600	32.0400	42.1000	32.0500	60.8950	87.8950
	Inconsistency in PyTorch for `cosine similarity`	2.9010	×	2.9480	×	×	**0.0010**	0.0300	0.0020	0.0033	0.0020	0.0038	0.0061
[36]	Invalid input exception in `Root Mean Squared`	0.0126	7.1700	0.0685	×	×	**0.0030**	0.0030	0.0030	0.0048	0.0030	0.0057	0.0092
[14]	NaN in `softmax`	×	×	×	4.2940	0.7060	**0.6920**	0.7040	0.8830	1.1000	0.7160	1.6777	2.6550
[34]	Incorrect value in `linear solver`	×	×	×	×	×	**1.0200**	×	1.4450	4.0200	2.7760	2.7455	×
[35]	Wrong prediction due to vanishing gradient	0.7069	32.4000	0.4121	×	×	**0.6260**	0.9340	0.9630	1.5020	0.9220	1.8285	2.8920
Total Bugs		3	3	3	3	3	**13**	11	13	13	13	13	12
Average Time		1.2068	13.2043	1.1432	54.3580	0.9213	**1.9182**	3.0555	4.0453	4.1412	4.6637	5.7662	8.4913

Soft Assertion Fuzzer found 13 bugs from 15 real-world programs, shown in Table 5, and found all the GRIST bugs (79 total, see the first row in Table 6). We listed the links of the GitHub repositories, and reported the failure symptoms in the first and second columns of Table 5. Column *SA Fuzzer* reports our results using the soft assertion trained with the 14×14 dataset. This dataset reported the best average accuracy in Table 4. Each tabular reports the time used to trigger the bug. If the bug is not found, we mark an ×. Since the GitHub bug is unknown, an × could mean that there does not exist a numerical bug in the application. Our results show that finding these bugs are very fast in seconds (1.92 seconds on average) using our SA Fuzzer. Under *Failure Symptoms*, we show

Table 6. Outperform the State-of-the-Art Techniques

Fuzzer	GRIST (79)	GRIST Average Time (sec)	Real-World (15)	NaN/INF	Other Failures
SA Fuzzer	79	0.646	13	88	4
RANUM [25]	79	2.209	3	82	0
GRIST [47]	78	44.267	3	81	0
Atheris [28]	25	0.283	3	27	1
PyFuzz [18]	24	5.498	3	26	1
Hypothesis [40]	23	5.945	3	25	1

that we can detect NaN/INF errors, as well as incorrect results. Under *Comparing with Baselines*, we listed 5 baseline results for the real-world bugs, which we will discuss in more details in Section 5.4.



Brain MRI showing visible tumor

```
Image: Y6.jpg
Incorrect Model Prediction: 0.4990
Correct Model Prediction: 0.7788
--------------------------------------
Image: Y194.jpg
Incorrect Model Prediction: 0.4991
Correct Model Prediction: 0.7499
--------------------------------------
Image: Y180.jpg
Incorrect Model Prediction: 0.4984
Correct Model Prediction: 0.8692
--------------------------------------
Image: Y90.jpg
Incorrect Model Prediction: 0.4985
Correct Model Prediction: 0.9064
--------------------------------------
Image: Y53.jpg
Incorrect Model Prediction: 0.4987
Correct Model Prediction: 0.8014
--------------------------------------
```

Prediction results before and after fix

Fig. 4. Numerical instability leads to model mispredictions

Case Study: Finding an unknown bug in a tumor detection model. One of the numerical instability we found is in TumorScope [35], an ML model trained for tumor detection in brain MRI images using Kaggle dataset [4] (see Table 5, the case with vanishing gradient [35]). Due to this bug, some images with tumor (shown the left in Figure 4) are incorrectly predicted with no tumor.

We observed that during inference, these input images triggered numerical instability in the `Conv2D` layer, shown at line 7 in Figure 5, and produced very small or negative values. The `ReLU` activation function, applied after the `Conv2D` layer, sets all negative values to zero. This results in many neurons outputting zero. As these zeros passed to the subsequent `MaxPool2D` layer (line 9), the layer again propagates mostly zeros. MaxPooling selects the maximum value from each window size of the feature map, so if most inputs to the MaxPooling operation are zero, the output will also be predominantly zeros. Therefore, at the dense layer (line 11), the features extracted from the image are mostly irrelevant, causing the model to produce an incorrect prediction in the final output layer (line 17).

We found that the numerical instability was triggered in `Conv2D` because it includes an unstable weight initialization function `GlorotUniform`; as a result, the weights are not properly initialized during training and lead to small/negative values in ReLU activations, causing classification errors. When we use a stable weight initialization algorithm, namely `he_normal`[20], and add it at line 7 in Figure 5, the numerical instability is then fixed. The model correctly classified the same tumor image on the left in Figure 4 with a confidence score of 0.8874 and also correctly predicted other images, shown on the right in Figure 4.

5.4 RQ3: Outperform the Baselines

In Table 6, we summarized our comparison with the baselines. Because we enabled automatic learning the error conditions in soft assertions, we supported a more variety of unstable functions; that allows us to more effectively find bugs in real-world code, shown in Table 5 under *Failure Symptoms*. Among the 15 real-world projects, *Soft Assertion Fuzzer* detected 13 bugs while other baselines all detected 3, shown in Table 6 under *Real-World*. We developed a comprehensive set of

```
import ..
x = load_and_preprocess_images(image_paths)
y = labels  # Labels (1 for tumor, 0 for no tumor)
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.33, shuffle=True)
def TumorScope():
    model = models.Sequential([layers.InputLayer((200, 200, 3)),
    layers.Conv2D(16, (3, 3), activation='relu'),
        ......
        layers.MaxPool2D(),
        ......
        layers.Dense(1, activation='sigmoid')])
    model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
    return model
TumorScope.fit(x_train, y_train, epochs=10, validation_split=0.2) # Train the model
TumorScope = TumorScope()
test_image = load_and_preprocess_image(image_path)
prediction = TumorScope.predict(test_image)[0][0] #Prediction
```

Fig. 5. TumorScope Model

oracles to capture the failure symptoms beyond *NaN/INF*. See the total failures and types generated under *NaN/INF* and *Other Failures* in Table 6 for both GRIST and real-world programs.

Our results also show that soft assertion guidance effectively helped trigger the bug. On the GRIST benchmark containing 79 known numerical bugs, *Soft Assertion Fuzzer* matched RANUM's [25] performance by detecting all 79 bugs (compared to GRIST's 78). Although GRIST and RANUM both can handle `exp`, only our tool found the INF bug in `exp` function reported in [10] in Table 5. We believe that the soft assertion guidance also helps quickly trigger the bugs. SA Fuzzer reported 0.646 seconds under *GRIST average time*.

5.5 RQ4: Ablation Study

We studied whether the use of history inputs (see Section 4.3) helped performance. We ran *Soft Assertion Fuzzer* by turning off the history information and reported the results under *No H* in Table 5 under *Ablation Study*. Our results show that without using history information, we only found 11 bugs, instead of 13, in real-world programs, and it took longer to find the bugs.

We also reported fuzzing results that used soft assertions trained with different sizes of input matrices. See Columns *3x3* and *MNIST* in Table 5. Interestingly, SA Fuzzer found the same 13 bugs in the real-world programs in these configurations. We observed that a soft assertion trained with 3×3 input matrices can handle ML applications that take the real-world datasets, such as MNIST, as input. We see that 14×14 configuration (under Column *SA Fuzzer*) reported the fastest time for most of the bugs. We believe that the efficiency is related to the model accuracy. When we have low mispredictions in soft assertions, the search of the input is faster. For GRIST, all the configurations of SA Fuzzer can trigger 79 bugs. For *No H*, 3×3 and *MNIST*, we reported time of 0.793, 0.721 and 0.767 seconds respectively (please see our replication package for details).

We further evaluated the impact of SA classifier architecture and training data size on the performance of the *Soft Assertion Fuzzer* . When the soft assertions are trained on 40K data points using `Random Forest` classifier, our SA Fuzzer detected all bugs with an average time of 1.9182 seconds. Replacing the random forest with a deep neural network (DNN) increased the detection time to 4.1412 seconds, making it 2.16 times slower without improving bug detection. Regarding training data size, the 40K model remained the most efficient, achieving an average time of 4.6637

seconds in the 3 × 3 configuration. Reducing the training data to 15K and 5K led to slower detection times of 5.7662 and 8.4913 seconds, respectively. The 5K model also missed one bug.

6 Threats to Validity and Discussions

When constructing oracles, we confirmed some of the unstable functions with the PyTorch developers. We performed code review and debugging to make sure our implementations of the oracles, the soft assertion generators and the fuzzer were correct. We used 94 ML applications consisting of the programs from GRIST as well as the real-world code from GitHub. We followed a clearly defined procedure (Section 5.1.3) to select programs to construct our benchmark. We trained soft assertions for 25 unstable functions covering a variety of categories. We used the 5 SOTA baselines for comparison. We ran the fuzzers 5 times to get average results. We also manually inspected the code to confirm the triggered bugs, one of which has been confirmed by the GitHub developers.

To explore the possibility of using traditional specification inference tools to infer safe input conditions for unstable functions, we tried the open-source tool Daikon [7]. The first challenge is that Daikon is not compatible with Python-based ML frameworks like TensorFlow and PyTorch. Thus we ported an unstable function `exp()` to a C++ implementation and ran unit tests to get the traces for Daikon. Daikon failed to report any invariants related to matrix elements; instead, it provides irrelevant properties such as `return>x` and many other trivial properties. In this case, the safe input condition is that every element in the input matrix should be less than 88.72 (see our oracle documentation in the replication package). We successfully generated a soft assertion for this function and detected the instability caused by this unstable function in ML applications.

7 Related Work

Numerical Bugs: GRIST [47] and our approach both aim to detect numerical bugs in ML codes. GRIST is also used automatic differentiation to determine input adjustments. Our novelty is that we used automatically generated soft assertions to guide the input search, while GRIST used manually-encoded safety/error conditions. As a result, we supported much more unstable functions where we cannot manually write safety/error checks. Also, we can find numerical bugs using 6 types of oracles, which allows us to find more types of numerical bugs, while GRIST used NaN/INF to detect failures. DEBAR [50] used abstract interpretation to statically detect numerical bugs. RANUM [25] combines GRIST and DEBAR and fixed the numerical bugs by clipping the values based the range reported by DEBAR. There are also empirical studies on numerical bugs [11, 21, 22, 39, 41, 42]. For example, DeepStability [24] studied numerical bugs in deep learning libraries. It built a database of numerical bugs, from which, we collected 56 unstable functions. We added more unstable functions and also designed 6 types of oracles and implemented it for 25 unstable functions.

Fuzzing for ML code and models: Fuzzing has been applied to ML applications [47], ML libraries [33, 48], ML models [16, 30, 32, 49], and deep learning compilers [26]. ΔFuzz [48] applied different execution scenarios as test oracles to test first- and high-order gradients targeting the Automatic Differentiation component in DL libraries. CRADLE [33] used differential testing and found the equivalent functions in the libraries as oracles for fuzzing. DRFuzz [49] found regression faults in ML models by generating inputs that maximize output discrepancies. DeepXplore [32] aimed neuron activation coverage to fuzz neural network models. DLFuzz [16] used mutation and differential testing to test DNN models aiming to achieve neuron coverage. NNSmith [26] used differential testing and gradient based search to find model inputs to test DNN compilers.

8 Conclusions

In this paper, we proposed a new concept *soft assertion* and developed *Soft Assertion Fuzzer* to find numerical bugs in ML applications. Our work is motivated by the fact that it is very hard, even for ML experts, to write `assert` or symbolic condition checks for numerical stability of ML code, due to the high dimensional input types such as tensors and complex math functions. The soft assertion is an automatically learned ML model that encodes the safety/error conditions for an unstable function. To generate soft assertions, we first built a database of unstable functions and constructed their testing oracles. We developed a novel approach of generating the training dataset by running the unit tests of unstable functions. During fuzzing, soft assertions provide signals to guide fuzzers to perform effective and targeted mutations. We performed a comprehensive evaluation on 94 ML applications and compared with 5 SOTA baselines. Our results show that soft assertions are very effective to guide fuzzing. We detected 92 numerical bugs, 13 of them are unknown bugs from the important real-world GitHub code, significantly outperformed the baselines. While all the baselines focus on reporting NaN/INF failures, we found numerical bugs that produced incorrect output. In the future, we will continue improving soft assertion accuracies and explore more applications of soft assertions.

9 Data Availability

Our replication package is located at https://figshare.com/s/6528d21ccd28bea94c32.

Acknowledgments

This research is partially supported by the U.S. National Science Foundation (NSF) under Award #2313054. We thank the anonymous reviewers for their valuable and insightful comments, which greatly improved our work. We also thank Ashwin Kallol Joshy for his helpful guidance and support in understanding the project during its early stage.

References

[1] BESBES Ahmed. 2021. focal_loss.py. GitHub. Available at: https://github.com/ahmedbesbes/character-based-cnn/blob/master/src/focal_loss.py. Accessed: 2024-03-21.

[2] Atilim Gunes Baydin, Barak A Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. 2018. Automatic differentiation in machine learning: a survey. *Journal of machine learning research* 18, 153 (2018), 1–43. https://jmlr.org/papers/volume18/17-468/17-468.pdf

[3] Richard L. Burden and J. Douglas Faires. 2001. *Numerical Analysis* (7th ed.). Brooks/Cole, Pacific Grove, CA.

[4] Navoneel Chakrabarty. 2024. Brain MRI Images for Brain Tumor Detection. GitHub. Available at: https://www.kaggle.com/datasets/navoneel/brain-mri-images-for-brain-tumor-detection/data. Accessed: 2024-09-10.

[5] Sangeeta Chowdhary, Jay P Lim, and Santosh Nagarakatte. 2020. Debugging and detecting numerical errors in computation with posits. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 731–746. doi:10.1145/3385412.3386004

[6] Sangeeta Chowdhary and Santosh Nagarakatte. 2021. Parallel shadow execution to accelerate the debugging of numerical errors. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 615–626. doi:10.1145/3468264.3468585

[7] codespecs. 2024. Daikon Documentation. GitHub. Available at: https://plse.cs.washington.edu/daikon/pubs/. Accessed: 2025-01-22.

[8] Rodrigo Pérez Dattari. 2023. ranking_losses.py. GitHub. Available at: https://github.com/rperezdattari/Stable-Motion-Primitives-via-Imitation-and-Contrastive-Learning/blob/main/src/agent/utils/ranking_losses.py. Accessed: 2024-03-21.

[9] Bruce Dawson. 2008. Comparing Floating Point Numbers, 2008 Edition. http://www.cygnus-software.com/papers/comparingfloats/comparingfloats.htm. Accessed: 2024-03-21.

[10] DeCLaRe. 2022. OT_solver.py. GitHub. Available at: https://github.com/declare-lab/MM-Align/blob/main/src/modules/OT_solver.py. Accessed: 2024-03-21.

[11] Anthony Di Franco, Hui Guo, and Cindy Rubio-González. 2017. A comprehensive study of real-world numerical bug characteristics. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 509–519. doi:10.1109/ASE.2017.8115662

[12] Yanjia Li Ethan. 2023. loss.py. GitHub. Available at: https://github.com/ethanyanjiali/minChatGPT/blob/main/src/loss.py. Accessed: 2024-03-21.

[13] Maryellen L Giger. 2018. Machine learning in medical imaging. *Journal of the American College of Radiology* 15, 3 (2018), 512–520. doi:10.1016/j.jacr.2017.12.028

[14] Ethan Gilmore. 2024. MicroMLP. GitHub. Available at: https://github.com/ethangilmore/MicroMLP. Accessed: 2024-09-10.

[15] Carlos A Gomez-Uribe and Neil Hunt. 2015. The netflix recommender system: Algorithms, business value, and innovation. *ACM Transactions on Management Information Systems (TMIS)* 6, 4 (2015), 1–19. doi:10.1145/2843948

[16] 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 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 739–743. doi:10.1145/3236024.3264835

[17] Nikita Gushchin. 2023. cunet. GitHub. Available at: https://github.com/ngushchin/EntropicNeuralOptimalTransport/tree/e66cf61b37f06f0714165161618f7a944af189c9. Accessed: 2024-09-10.

[18] Fabrice Harel-Canada. 2022. A basic python-based fuzzing tool used to support miscellaneous research objectives. GitHub. Available at: https://github.com/fabriceyhc/pyfuzz. Accessed: 2023-15-11.

[19] Richard Hawkins, Colin Paterson, Chiara Picardi, Yan Jia, Radu Calinescu, and Ibrahim Habli. 2021. Guidance on the Assurance of Machine Learning in Autonomous Systems (AMLAS). *arXiv preprint arXiv:2102.01564* (2021). doi:10.48550/arXiv.2102.01564

[20] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*. IEEE Computer Society, 1026–1034. doi:10.1109/ICCV.2015.123

[21] Sharon Chee Yin Ho, Vahid Majdinasab, Mohayeminul Islam, Diego Elias Costa, Emad Shihab, Foutse Khomh, Sarah Nadi, and Muhammad Raza. 2023. An empirical study on bugs inside PyTorch: A replication study. (2023), 220–231. doi:10.1109/ICSME58846.2023.00031

[22] Li Jia, Hao Zhong, Xiaoyin Wang, Linpeng Huang, and Xuansheng Lu. 2020. An empirical study on bugs inside tensorflow. In *Database Systems for Advanced Applications: 25th International Conference, DASFAA 2020, Jeju, South Korea, September 24–27, 2020, Proceedings, Part I 25*. Springer, 604–620. doi:10.1007/978-3-030-59410-7_40

[23] Jinwoo Kim. 2023. SO.py. GitHub. Available at: https://github.com/jw9730/lps/blob/b6ba12faa9ec74ba95d95299895a21da944dd4af/src/symmetry/groups/SO.py. Accessed: 2024-03-21.

[24] Eliska Kloberdanz, Kyle G Kloberdanz, and Wei Le. 2022. DeepStability: A study of unstable numerical methods and their solutions in deep learning. In *Proceedings of the 44th International Conference on Software Engineering*. 586–597. doi:10.1145/3510003.3510095

[25] Linyi Li, Yuhao Zhang, Luyao Ren, Yingfei Xiong, and Tao Xie. 2023. Reliability Assurance for Deep Neural Network Architectures Against Numerical Defects. 1827–1839 pages. doi:10.1109/ICSE48619.2023.00156

[26] Jiawei Liu, Jinkun Lin, Fabian Ruffy, Cheng Tan, Jinyang Li, Aurojit Panda, and Lingming Zhang. 2023. Nnsmith: Generating diverse and valid test cases for deep learning compilers. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 530–543. doi:10.1145/3575693.3575707

[27] Weitang Liu. 2023. triplet_loss.py. GitHub. Available at: https://github.com/lonePatient/TorchBlocks/blob/445140f8190a037b590d3344cdbd6cae5b850da0/src/torchblocks/losses/triplet_loss.py. Accessed: 2024-03-21.

[28] Google LLC. 2023. Atheris: A Coverage-Guided, Native Python Fuzzer. https://github.com/google/atheris Accessed: 2023-12-03.

[29] longyahui. 2021. crf.py. GitHub. Available at: https://github.com/longyahui/GCNMDA/blob/master/src/crf.py. Accessed: 2024-03-21.

[30] Augustus Odena, Catherine Olsson, David Andersen, and Ian Goodfellow. 2019. TensorFuzz: Debugging Neural Networks with Coverage-Guided Fuzzing. In *Proceedings of the 36th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 97)*, Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.). PMLR, 4901–4911. doi:10.5555/3327757.3327811

[31] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic Differentiation in PyTorch. In *NIPS 2017 Workshop on Automatic Differentiation*.

[32] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. 2017. DeepXplore: Automated Whitebox Testing of Deep Learning Systems. (2017), 1–18. doi:10.1145/3132747.3132785

[33] 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 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). *IEEE Computer Society, Los Alamitos, CA, USA* (2019), 1027–1038. doi:10.1109/ICSE.2019.00107

[34] Robin Scheibler. 2023. diffusion-separation. GitHub. Available at: https://github.com/fakufaku/diffusion-separation/blob/main/utils/linalg.py. Accessed: 2024-09-10..

[35] Mohit Shaharwale. 2024. TumorScope. GitHub. Available at: https://github.com/mohits2806/TumorScope. Accessed: 2024-09-10.

[36] Simon. 2024. Value Error. GitHub. Available at: https://github.com/scikit-learn/scikit-learn/issues/29678. Accessed: 2024-09-07.

[37] Strath.AI. 2023. SatelliteCloudGenerator. GitHub. Available at: https://github.com/strath-ai/SatelliteCloudGenerator/blob/d120eaf787d379527d7aeb4f7cb3f635b00b50d5/src/noise.py. Accessed: 2024-03-21.

[38] Zeye Sun. 2023. loss.py. GitHub. Available at: https://github.com/sunzeyeah/RLHF/blob/cd1a6d54971eb0513f38974aa6dcca53aa2f3174/src/models/loss.py. Accessed: 2024-03-21.

[39] Florian Tambon, Amin Nikanjam, Le An, Foutse Khomh, and Giuliano Antoniol. 2024. Silent Bugs in Deep Learning Frameworks: An Empirical Study of Keras and TensorFlow. 10 pages. doi:10.1007/s10664-023-10389-6

[40] Hypothesis Development Team. 2023. Hypothesis: A new approach to property-based testing. https://hypothesis.readthedocs.io/en/latest/ Accessed: 2024-01-26.

[41] Jackson Vanover, Xuan Deng, and Cindy Rubio-González. 2020. Discovering discrepancies in numerical libraries. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 488–501. doi:10.1145/3395363.3397380

[42] Gan Wang, Zan Wang, Junjie Chen, Xiang Chen, and Ming Yan. 2022. An empirical study on numerical bugs in deep learning programs. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–5. doi:10.1145/3551349.3559561

[43] Qiang Wang, Thanh-Tung Nguyen, Joshua Z Huang, and Thuy Thi Nguyen. 2018. An efficient random forests algorithm for high dimensional data classification. *Advances in Data Analysis and Classification* 12 (2018), 953–972. doi:10.1007/s11634-018-0318-1

[44] Cody Watson, Michele Tufano, Kevin Moran, Gabriele Bavota, and Denys Poshyvanyk. 2020. On learning meaningful assert statements for unit test cases. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 1398–1409. doi:10.1145/3377811.3380429

[45] Robert White and Jens Krinke. 2020. Reassert: Deep learning for assert generation. *arXiv preprint arXiv:2011.09784* (2020). doi:10.48550/arXiv.2011.09784

[46] Li Xingxuan. 2023. prior_wd_optim.py. GitHub. Available at: https://github.com/DAMO-NLP-SG/MVCR/blob/main/src/prior_wd_optim.py. Accessed: 2024-03-21.

[47] 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 Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 627–638. doi:10.1145/3468264.3468612

[48] Chenyuan Yang, Yinlin Deng, Jiayi Yao, Yuxing Tu, Hanchi Li, and Lingming Zhang. 2023. Fuzzing automatic differentiation in deep-learning libraries. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1174–1186. doi:10.1109/ICSE48619.2023.00105

[49] Hanmo You, Zan Wang, Junjie Chen, Shuang Liu, and Shuochuan Li. 2023. Regression Fuzzing for Deep Learning Systems. (2023), 82–94. doi:10.1109/ICSE48619.2023.00019

[50] Yuhao Zhang, Luyao Ren, Liqian Chen, Yingfei Xiong, Shing-Chi Cheung, and Tao Xie. 2020. Detecting numerical bugs in neural network architectures. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 826–837. doi:10.1145/3368089.3409720

Received 2025-02-25; accepted 2025-04-01