Predoo: Precision Testing of Deep Learning Operators

Xufan Zhang
State Key Laboratory for Novel Software Technology at Nanjing University, China
xufan.zhang@outlook.com

Ning Sun
State Key Laboratory for Novel Software Technology at Nanjing University, China
MF20320131@smail.nju.edu.cn

Chunrong Fang*
State Key Laboratory for Novel Software Technology at Nanjing University, China
fangchunrong@nju.edu.cn

Jiawei Liu
State Key Laboratory for Novel Software Technology at Nanjing University, China
eudemoniajw@gmail.com

Jia Liu
State Key Laboratory for Novel Software Technology at Nanjing University, China
liujia@nju.edu.cn

Dong Chai
HiSilicon, Huawei, China
chaidong@hisilicon.com

Jiang Wang
HiSilicon, Huawei, China
jiang.wang@hisilicon.com

Zhenyu Chen
SKL, Nanjing University, China
zychen@nju.edu.cn

ABSTRACT

Deep learning(DL) techniques attract people from various fields with superior performance in making progressive breakthroughs. To ensure the quality of DL techniques, researchers have been working on testing and verification approaches. Some recent studies reveal that the underlying DL operators could cause defects inside a DL model. DL operators work as fundamental components in DL libraries. Library developers still work on practical approaches to ensure the quality of operators they provide. However, the variety of DL operators and the implementation complexity make it challenging to evaluate their quality. Operator testing with limited test cases may fail to reveal hidden defects inside the implementation. Besides, the existing model-to-library testing approach requires extra labor and time cost to identify and locate errors, i.e., developers can only react to the exposed defects. This paper proposes a fuzzing-based operator-level precision testing approach to estimate individual DL operators' precision errors to bridge this gap. Unlike conventional fuzzing techniques, valid shape variable inputs and fine-grained precision error evaluation are implemented. The testing of DL operators is treated as a searching problem to maximize output precision errors. We implement our approach in a tool named Predoo and conduct an experiment on seven DL operators from TensorFlow. The experiment result shows that Predoo can trigger larger precision errors compared to the error threshold declared in the testing scripts from the TensorFlow repository.

*Chunrong Fang is the corresponding author of this paper.

ISSTA '21, July 11–17, 2021, Virtual, Denmark

ACM ISBN 978-1-4503-8459-9/21/07…$15.00
https://doi.org/10.1145/3460319.3464843

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; • **Mathematics of computing** → *Numerical analysis*; • **Computing methodologies** → *Neural networks*.

KEYWORDS

precision testing, deep learning operators, floating-point operation

ACM Reference Format:
Xufan Zhang, Ning Sun, Chunrong Fang, Jiawei Liu, Jia Liu, Dong Chai, Jiang Wang, and Zhenyu Chen. 2021. Predoo: Precision Testing of Deep Learning Operators. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '21), July 11–17, 2021, Virtual, Denmark.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3460319.3464843

1 INTRODUCTION

Deep learning(DL) techniques boost research progress in various fields including computer vision [22], medical diagnosis [2], and even software testing [43]. Developers make use of available DL libraries to build, train and deploy DL applications. Multiple DL libraries grant developers more direct access to DL techniques on devices with different computation capabilities. Mature DL applications are expected to show competitive and stable performance in handling tasks. However, researchers have found that these applications could be vulnerable to small perturbations [40]. According to some empirical studies [41] [17], DL libraries could also raise defects in applications. Since developers know little about DL libraries' implementation, it becomes challenging to avoid such risks.

As an effective form of machine learning [10], DL techniques are integrated in more DL libraries, including TensorFlow [1], PyTorch [27], and the Microsoft Cognitive Toolkit(CNTK) [39]. Meanwhile, new DL libraries like MindSpore and MNN [18] keep being developed to extend use scenarios, ranging from high-performance servers to resource-limited IoT devices. In DL libraries, algorithms for processing multi-dimensional inputs, i.e., tensors, are implemented as callable DL operators. The DL operator discussed in this paper is essentially an API call to perform tensor manipulations.

When performing an inference task, these operators are responsible for processing and transforming tensors to produce the output. Since scenarios offer different computing capabilities, e.g., data representation ability and hardware performance, such differences could introduce computational errors to operator behaviors. Most DL operators perform complex and non-linear transformations, which often cause precision-related errors [8]. Besides, library developers keep optimizing the operator's implementation, e.g., reducing iteration times of Newton's method, to compute faster.

One of the main challenges in testing numerical operators is to provide the testing oracle. The precision error makes it hard to determine whether the produced output is correct because it might fluctuate. Comparing the output with the result produced under infinite precision, i.e., absolute error, is an option to solve it. Because it is impractical to provide the result of infinite precision in automatic testing, the output from a higher-precision instance is used instead [12]. Correspondingly, absolute tolerance(*atol*) and relative tolerance(*rtol*) are metrics that widely adopted to evaluate precision errors. However, it remains a problem to give these metrics appropriate values. According to our study on testing practices from library developers, different values are used when testing different operators, indicating that tolerance concretization requires expert knowledge about the operator. An alternative solution followed by library developers is to collect the output from an equivalent program that serves the same functionality for reference and quantify the difference, i.e., relative error [26].

Another challenge in testing DL operators is to generate sufficient test input. Unlike other primitive data types, tensor holds a rather complex structure in its shape. Library developers prepare at most dozens of tensor inputs in the testing scripts. However, these hand-written tests' effectiveness is limited, considering the test input scale and output evaluation. Some researchers overcome this problem by collecting intermediate output when investigating DL models. As enough test inputs for the DL model are provided, they can collect sufficient test data for each operator inside the model. By detecting the DL model's misbehavior and tracing inconsistency patterns, researchers manage to locate defects in the underlying operators [30]. However, from the perspective of software engineering, DL libraries should first be packaged and released, then model developers can have access to these encapsulated operators. Therefore, model construction is a practice out of the library development lifecycle. This kind of model-to-library approach requires extra cost for model construction and intermediate output extraction.

To address the challenges mentioned above, we introduce a fuzzing-based precision testing approach and implement it in a tool named Predoo to maximize precision error found on DL operators. Predoo extends numerical precision testing to DL operators, which processes the more complex tensor input. Mutation methods are introduced to implement mutation-based input generation, which helps provide many test inputs. Unlike conventional fuzzing techniques that generate semi-valid test inputs to trigger exceptions or crashes [34], Predoo follows the strict shape constraint of DL operators to generate valid tensors. Also, precision errors are different from exceptions or crashes, i.e., a DL operator can produce an output, although it is hard to tell its correctness. To improve test efficiency, Predoo implements three guiding metrics. With these metrics, Predoo prioritizes inputs that trigger larger errors. To evaluate the effectiveness and efficacy of Predoo, we experiment on seven DL operators. We compare errors found by Predoo against the built-in threshold and CRADLE. The result shows that Predoo can trigger precision errors one order of magnitude larger on average. Predoo is the first precision testing work that directly investigates DL operators to the best of our knowledge. The main contributions of this paper are as follows:

1) study current test implementation from DL libraries to summarize test practices and goals in operator testing, including the preparation of input and the evaluation of output. Furthermore, we point out limitations and potential risks.
2) propose a precision-error-guided fuzzing approach to test DL operators. We refine the precision-related metrics in evaluating tensor outputs, with which we demonstrate the ability of our approach to finding error-inducing inputs.
3) conduct a comprehensive experiment on seven widely used DL operators from TensorFlow and compare the result against the built-in threshold and CRADLE to illustrate its efficacy and efficiency.
4) public the source code of this tool and release our test data to help other researchers conduct relevant numeric error analysis in DL libraries.

2 BACKGROUND

Precision-related errors caused by floating-point computation approximation may impact the functionality of an application, especially for numerical computation programs. According to a study on numerical software libraries [9], numerical bugs account for 32% of all the examined bugs. To figure out the impacts of floating-point values, researchers work on approaches to estimate round-off errors [33], identify maximized errors [12] and capture arithmetic exceptions [21] by employing verification and testing techniques. Operators in DL libraries face similar challenges because they use non-linear algorithms to process the more complex shape-variable inputs. Besides, DL operators allow practitioners to customize input precision and operator precision. The input precision indicates the precision of elements inside the tensor, while the operator precision determines which precision the operator uses to store variables and finish the computation. For DL applications, detailed implementation considers the tradeoff between performance and precision. However, it remains unclear about the influence.

2.1 Precision Testing

In precision testing, error implies the difference between the produced output and the expected value, i.e., test oracle. Unless otherwise specified, the error mentioned in this paper refers to precision error. Formal error analysis can be complicated and requires expert knowledge about the implemented operator, which guarantees an operator's safety. Meanwhile, the inherent computational error brought by finite-bit representation in computer programs raises people's concern about its impact, especially for floating-point numerical computation programs. Due to such errors' uncertainty and unpredictability, it is difficult to determine an output's correctness. As a walkaround, researchers put forward metrics that help estimate whether the disturbance in the output is within an

acceptable range, i.e., errors can be bounded [16]. Formally, it can be described as $\mid P(i) - O_i \mid < \varepsilon$, where P denotes the numerical program, i denotes an input, O_i denotes the expected output for input i, and ε denote the acceptable range. Input i generation, oracle O_i approximation, and error bound ε definition to become three tasks in precision testing.

Input generation. Random testing [14] provides random, independent inputs to a program. The chance of hitting error-prone inputs depends on the magnitude of the defect rate. In consequence, it could be inefficient to find an error-inducing input. To bridge this gap, adaptive random testing is proposed. Adaptive random testing selects the farthest input from executed test inputs. However, random testing techniques cannot activate the profound logic because it is a structure unaware. Symbolic execution [32] takes program implementation into account when generating a new input. Though it could be efficient in exposing different execution paths, it requires additional access to the source code. Meanwhile, extra time will be spent analyzing variables and data structures.

Oracle approximation. There are two kinds of test oracles, implicit and explicit [5]. Implicit test oracles, including runtime exceptions and crashes, do not require domain-specific knowledge to distinguish between correctness and incorrectness. For a numerical program, such anomalies are sufficient to prove the presence of critical defects. However, implicit test oracles cannot reveal correctness-related issues. To achieve this, researchers make use of alternative yet independent programs or multiple version programs to provide explicit oracles, i.e., differential testing [24] or regression testing [36]. Let $P^{'}$ denote such a function-equivalent program, formally we have $P^{'} \neq P$, while $P^{'}(i) = P(i)$ is expected. Unfortunately, for numerical programs, even if we find such a $P^{'}$, directly judging whether the output is equal is not applicable.

Error bound definition. An appropriate error bound ε helps identify fine-grained value check between $P^{'}(i)$ and $P(i)$. In practice, the external environment and internal implementation make it a complicated task to determine ε. The external environment limits the capacity of finite-bit representation in implementation, impacting the propagation of the precision error. Thus there are no universal criteria to define the error bound. If the configured ε is too large, it cannot expose potential defects. On the contrary, if it is too small, time would be spent on non-serious problems. Therefore, instead of explicitly defining the error bound ε, researchers transform it into a searching problem [44]. Let $\mathbb{I}$ denote the input space, $\mathbb{T}$ denote the test set produced by the searching strategy $\mathbb{S}$, $\mathbb{T} \subseteq \mathbb{I}$, $\mathcal{E}_i$ denote the error for input i, $i \in \mathbb{T}$, the proposed approach $\mathbb{S}$ aims to find input α that can maximize precision error.

$$\mathbb{T} = \mathbb{S}(P, \mathbb{I}), \forall i \in \mathbb{T}, \mathcal{E}_\alpha = max(\mathcal{E}_i)$$

Following this idea, most precision testing works on effective strategies $\mathbb{S}$ to explore the input space, to maximize error.

2.2 Deep Learning Operators

DL operators provide vital support for DL-related tasks because DL models rely on them to perform concrete learning and inference tasks [7]. Operators need to update the hyper-parameter during the training phase, and the output is commonly considered some probability distribution. If the intermediate output changes, the result given by the DL model can be different. However, as the floating-point operation is implemented in the DL libraries, it is difficult for a model developer to tell whether the DL operator's output is sufficient to cause the difference in the final result. Meanwhile, the external environment could be different during training and deployment, where support from underlying hardware differs. We investigate three common categories of DL operators widely used in constructing DL models.

The **convolution** operator is efficient in describing transformations that apply the same linear sub-region transformations since it requires much less floating-point operations than the straightforward matrix multiplication algorithm. However, it still requires a lot of floating-point operations to implement the arithmetic. For a CNN [20] processing the image classification task, if a convolution kernel contains 4 elements (2×2), and the size of the corresponding output is 28×28, then $28 \times 28 \times 5 = 3920$ floating-point operations are required to implement this operator. As is mentioned above, floating-point operations could be error-prone. Thus the feature map represented by the output of the convolution operator should be carefully considered. Besides, there could be multiple implementations for a single DL operator in the same DL library, and each of them is independent of others. For example, in TensorFlow, there are approximately ten individual code implementations related to *conv2d* using different processors, e.g., CPU, GPU, and various data types, e.g., float16, float32, float64, etc.

The **pooling** operator helps to aggregate the statistics of the features at nearby locations, which helps judge whether an expected feature is present or not. Similar to the convolution operator, the pooling operator amplifies features in a small local area. Nevertheless, it does not keep a record of the accurate location of it. For example, if the *max_pooling* operator is employed, it will assign the maximum value to all elements in a rectangular neighborhood in implementation. Intuitively, the operator does not care much about each element's accuracy since it prefers the maximum value shown by a local region of the output.

Other **nonlinear** operators like *ReLU* [38], *sigmoid*, *tanh*, *softmax*, also aim to adapt with variety of data. Comparing to linear operators, these non-linear operators work to map the input to a particular range. For example, the *sigmoid* operator produces output between the range $[0, 1]$.

2.3 Motivation

DL library testing attracts researchers because recent work indicates that some DL model defects reside in components of the underlying DL libraries, i.e., DL operators. To better explain our motivation and illustrate our approach, we collect operator testing scripts, 353 in total, from TensorFlow, a popular DL library[1]. To better understand current testing practices, we study operator testing implementation from DL library developers from input generation and output estimation perspectives. These findings are discussed with and confirmed by our enterprise collaborators, who are also library developers. We summarize existing limitations in testing DL operators and explain our intuition in building a fuzzing-based DL operator testing approach.

Test inputs preparation. DL operators do not share much similarities in the input parameter list except that they process tensors,

[1] https://github.com/tensorflow/tensorflow/tree/master/tensorflow/python/kernel_tests

e.g. ***conv2d*** takes additional parameters including *height*, *strides*, *padding*, and *filter*, while ***relu*** takes no additional parameters. Unless otherwise specified, the input refers to the tensor input a DL operator takes. Other parameters are declared using primitive data types, e.g., *integer*, *string*. Testing techniques for these primitive data types are relatively mature, but they cannot process shape-variable tensor inputs. According to the study, DL library developers construct several or dozens of inputs by either declaring a fixed batch of random data or manually providing them.

Test outputs estimation. Inspired by the study of oracle approximation in [26], we try to find out metrics library developers use to determine the correctness of a DL operator. Most DL operators make use of assertions to check whether the test case should pass. Shape check method *assertShapeEqual* and value check method *assertAllClose* are two main assertions in estimating the correctness. DL library developers also provide specified *atol* and *rtol*. Formally, the result is expected to satisfy the following constraint.

$$| P^{'}(i) - P(i) | < rtol * | P^{'}(i) | + atol$$

According to the study, library developers are more likely to customize the tolerance value. Even for the same operator, library developers could specify different tolerance value when testing with different finite-bit representation, i.e., *float32*, *float64*. An appropriate setting of the tolerance value may require expert knowledge of a DL operator. Meanwhile, current tolerance is used to check element-wise value errors.

Besides, we find DL operator testing practices that are different from that of conventional numerical program testing.

1) DL operators can take format-variable tensors as inputs. e.g., ***conv2d*** can take tensors in *channel_first* format, i.e., NCHW, as well as in *channel_last* format, i.e., NHWC. In consequence, it is difficult to define and calculate distance between test inputs, which makes adaptive random testing not applicable.
2) DL operators can take inputs represented by different finite-bit length, i.e., for floating-point values, the input can be of *float16*, *float32*, and *float64*.
3) DL operators can store variables and process intermediate computation with values represented by different finite-bit length, i.e., DL operators can use *float16*, *float32*, *float64* to store and process variables.
4) DL operators can run on different hardwares, e.g., CPU and GPU. The implementation is different. Some testing practices fail to take hardware variety into consideration, e.g., ***conv2d*** is configured to always run in GPU mode if possible.

Model developers rely on these DL operators to implement model construction, model training, model inference, and model deployment. Anomalies observed during these activities are reported to the DL library development team by model developers. Inspired by such a practice, some recent work follows the model-to-library routine to identify and locate operator defects [30] [13] [23]. An empirical study on TensorFlow [41] also indicates that library-related issues are the main reason for bugs in DL applications. From the perspective of software engineering, operator-level testing can be done by library developers. If a defect is found, library developers can fix it before a release, saving model developers from running into these defects when constructing their DL models.

3 APPROACH

To address the issues mentioned above, we are motivated to 1) implement a generic approach to test various DL operators; and 2) propose a more direct approach to implement operator-level testing. We propose Predoo, a fuzzing-based approach to implement precision testing for DL operators. Unlike conventional fuzzing techniques, Predoo generates valid inputs instead of semi-valid inputs considering the strict shape constraint of the input. When evaluating the output, Predoo uses precision error instead of crashes to implement fine-grained analysis for DL operators.

Figure 1 illustrates its workflow. Specifically, a shape resolver is introduced to define the input shape of a DL operator. Together with the declared input size, the initial seed can be concretized. To generate more test inputs, we implement three mutation strategies to generate new test inputs by adding precision perturbations to the candidate input. Meanwhile, we introduce testing preparation considering the variable finite-bit representation characteristic for DL operators. To further improve its efficiency, we also implement three strategies to guide the testing process. Predoo is designed as a generic solution to the mentioned limitations in DL operator testing. Therefore, Predoo is applicable for various DL operators, requiring no injection to an operator's source code.

Algorithm 1 explains the implementation details of Predoo. The algorithm returns the maximum precision error ε that could be found. To achieve this, there are some parameters required.

- *shape* defines the structure of the tensor input. e.g., if the *shape* is defined as [1, 28, 28, 3], then input will be concretized as a four-dimension tensor with the given shape. Since DL operators can process input in different shapes, Predoo allows the tester to declare the required shape.
- *size* defines the initial size of the seeds. Seeds are provided as the original input in Predoo. The *size* parameter is used to control the scale of seeds.
- *tn* defines the termination condition. Predoo keeps generating new inputs until the scale of executed test inputs reaches the limitation given by *tn*.
- *ops* holds multiple instances for the same operator. Predoo holds instances for the same operator using different data types. e.g., *conv2d(dtype=float16)*, *conv2d(dtype=float32)*, *conv2d(dtype=float64)*.
- *trans* indicates the alternative format that the operator can process. Some DL operators can process tensors of a different format, e.g., *conv2d* can process tensors in both NCHW and NHWC formats, *trans* helps transpose the original input to the alternative format.
- *strat* indicates the guiding metrics during testing. We implement three strategies, i.e. *random*, $l_\infty error$, and $l_1 error$.
- *op_params* includes all other parameters required by the operator. Since these parameters are of primitive data types that can be tested with existing techniques, Predoo does not generate inputs for these parameters. Instead, *op_params* is provided with concrete values.
- *mode* has two optional values, i.e., ***input_mode*** and ***op_mode***. It determines which component the current task focuses on. The ***input_mode*** changes precision of the input to the operator, while the ***op_mode*** changes operator precision.

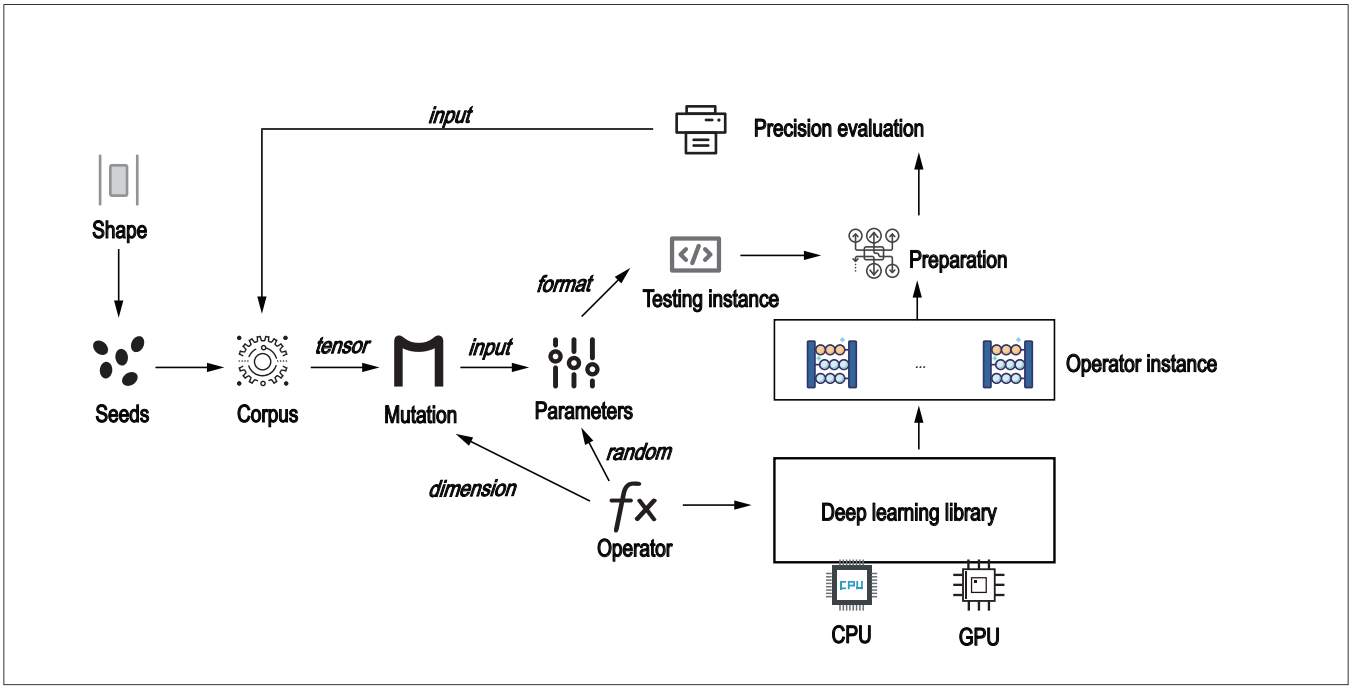


Figure 1: Predoo workflow

Among these parameters, *size*, *tn* are basic parameters to run the fuzzing task. To adapt fuzzing to DL operators, we introduce textitshape, *ops*, *trans* and *op_params*. We do not detail the above parameters because they will not change during the execution. *strat* and *mode* control the fuzzing process. We try to figure out their influences in the following section.

3.1 Seed Maintenance

Seeds are randomly generated in the set S of limited size *size* as initial test inputs. Since most DL libraries use multi-dimensional array [19] to represent tensors, we analyze tensors in this data format and will not talk about other representation methods like multi-linear maps [28]. Predoo requires external knowledge on the shape of the input to concretize the initial values in the test set S, e.g., $shape = [1, 28, 28, 3]$. Each seed is a shaped-array of floating-point samples from the standard normal distribution with the given *shape*. Each value inside a tensor input is floating-point values randomly generated. Without any prior knowledge about the DL operator under test, each value keeps the same probability of being picked.

With the provided parameter *size*, Predoo can generate an initial fixed-size test input set. When loaded into the corpus, these seeds become the original inputs to generate new test inputs, which will be added to the test set $\mathbb{T}$. In Predoo, picked inputs will not be put back to the input corpus. Therefore, if the corpus becomes empty before the termination, new seeds will be generated similarly.

3.2 Input Mutation

Mutation methods are responsible for generating new test inputs to run the operator under test [25]. Effective mutation methods contribute to generating valid inputs that expose higher precision errors inside the operator. Formally, we generate new test input by addding perturbations to the original input chosen from the corpus, i.e. $i^{'} = i + \delta$. In Predoo, δ is implemented by declaring the value range θ, then each element inside δ is in $[0, \theta]$. All values in the perturbation tensor are identical. In this way, we try to avoid coincidental impact from the introduced perturbation. δ is a tensor with the same shape as the original input. The value range of each element inside δ is limited. Therefore, the shape constraint of the input tensor will not break, and the generated input is valid. Predoo uses $\theta \in \{1e\text{-}4, 1e\text{-}6, 1e\text{-}8\}$ to implement mutation of different magnitude. This setting ensures that the change to the input is always small. All these values are smaller than the default tolerance for *float16*, i.e., 1e−3, the lowest precision we experiment in the evaluation.

Unlike random, mutation methods provide some heuristics in what changes testers want to apply to the original input. δ can be interpreted as a direction tensor. Therefore the newly generated input tries to explore the surrounding space. This is helpful considering the Lipschitz continuity property of DL operators [31]. Besides, since all the values in the perturbation tensor are identical, testers can control the direction of those changes.

3.3 Testing Preparation

Unlike conventional numerical programs, DL operator's main characteristic is that they are designed to use different precision to process data of different precision. To quantify errors caused by input precision changes or operator precision changes, Predoo allows testers to specify the current testing task's focus with *mode*. If ***input_mode*** is activated, the generated test inputs will be converted

Algorithm 1: predoo

Result: maximum error ε
Input: input shape: *shape*, seed_size: *size*, terminate_num: *tn*, operator: *ops*, transpose: *trans*, strategy: *strat*, parameters: *op_params*, mode: *mode*

```
count = 0, output_list = {} ;
𝕋 = init(shape, size) ;
corpus = load(𝕋) ;
while min < tn do
    candidate_input = choose(corpus);
    mutate_input_list = mutate(candidate_input);
    ordinary_params = random(op_params) ;
    try:
        switch mode :
            case input_mode :
                input_list = prepare(mutate_input_list) ;
                op_list = ops ;
            end
            case op_mode :
                input_list = mutate_input_list ;
                op_list = prepare(ops) ;
            end
            default :
                input_list = mutate_input_list ;
                op_list = ops ;
            end
        end
        output_list = execute(op_list, input_list,
          ordinary_params) ;
        error = relative_error(output_list, strat) ;
        if ε < error then
            𝕋 = load(input_list) ;
            increase(count) ;
            ε = error
        end
    catch Exception:
        // record a crash ;
    end
    return ε ;
end
```

to different data types. More specifically, for any input i, it will be converted to *float16*, *float32*, and *float64*. These prepared inputs will be used to run the provided operator instances. We analyze the impact of input precision changes on precision errors.

Similarly, the DL operator will be configured to run with different data types when ***op_mode*** is activated. i.e., for any operator P, the ***op_mode*** produces three operator instances running with *float16*, *float32*, and *float64* respectively. All these operator instances will be used as $P^{'}$ to calculate the observed relative error. We also study the impact of operator precision changes on precision errors, considering the diverse usage scenario of DL operators.

3.4 Error Analysis

In previous subsections, we illustrate how Predoo works to generate new valid test inputs and implement precision changes on the input or the operator. However, how a DL operator reacts to such changes remains unclear. Besides, model developers may use an operator multiple times in constructing a model, which gives us the intuition of running the DL operator multiple times. We try to demonstrate the infeasibility of threshold configuration for different DL operators, which helps us explain why Predoo finds higher precision errors instead of using a threshold for evaluation. Therefore, we theoretically analyze how the error is propagated and accumulated inside an operator on three categories of DL operators, i.e., convolution operators, pooling operators, and other non-linear operators.

Convolution operators. For simplicity, let K denote the kernel, I denote the input, the two-dimensional discrete convolution can be defined as follows.

$$C(i,j) = (K * I)(i,j) = \sum_p \sum_q I(i+p, j+q)K(p,q)$$

Let ϵ denote the absolute error on the input, where $\epsilon \ll I, \epsilon \ll K$, $\tilde{C}(i,j)$ denote the actual value, then we have:

$$\begin{aligned} \tilde{C}(i,j) &= \sum_p \sum_q (I(i+p, j+q) - \epsilon)(K(p,q) - \epsilon) \\ &= C(i,j) - \sum_p \sum_q (K+I)\epsilon + \sum_p \sum_q \epsilon^2 \\ &\approx C(i,j) - \sum_p \sum_q (K+I)\epsilon \end{aligned} \tag{1}$$

If a convolution operator is executed once, the difference between the real value $C(i,j)$ and the actual value $\tilde{S}(i,j)$ is as follows.

$$C(i,j) - \tilde{C}(i,j) = \sum_p \sum_q (K+I)\epsilon$$

Since we do not change other parameters of the operator during the testing preparation, i.e., p, q, K remain the same. If it runs n times, we have:

$$C_n(i,j) - \tilde{C}_1(i,j) \approx (pq)^n Kn\epsilon$$

As n increases, comparing to the error ϵ on original input, the error increases quickly. We can come to this conclusion for convolution on other dimensions in a similar way. Thus, the preparation stage is helpful in testing convolution operators.

Pooling operators. Let I denote the input, A denote the pooling area. For max pooling, we have:

$$P(i,j) = \max_{(p,q) \in A_{i,j}} I(p,q)$$

Similarly, let ϵ denote the absolute error on the input, where $\epsilon \ll I$, $\tilde{P}(i,j)$ denote the actual value, then we have:

$$\begin{aligned} \tilde{P}(i,j) &= \max_{(p,q) \in A_{i,j}} (I(p,q) - \epsilon) \\ &= \max_{(p,q) \in A_{i,j}} I(p,q) - \epsilon \\ &= P(i,j) - \epsilon \end{aligned} \tag{2}$$

As we can conclude from Equation 2, the pooling operator does not amplify the original error on the input. We can get similar conclusions for the average pooling operator. The error propagation scales linearly with the number of execution times. Although the pooling operator also returns a tensor as output, which can be used as an input for further iteration, the input change is of the pooling operator is less sensitive than the convolution operator.

Other non-linear operators. We further investigate on several common non-linear operators, e.g. *ReLU*, *sigmoid*, *tanh*, *softmax*.

ReLU. *ReLU* returns element-wise $max(x, 0)$. Let I denote the input, $x \in I$, ϵ_x denote the correspondent error for value x, $\tilde{ReLU}(x)$ denote the actual value, then we have:

Suppose $\epsilon > 0$, then for $x \geq \epsilon$, we have:

$$\tilde{ReLU}(x) = x - \epsilon_x = ReLU(x) - \epsilon_x \tag{3}$$

Similar to the max-pooling operator's proof, the error scales linearly with the number of execution times when $x \geq \epsilon_x$.

For $x \leq 0$, we have:

$$\tilde{ReLU}(x) = 0 = ReLU(x)$$

We can conclude that the *ReLU* operator is not affected by the error when $x \leq 0$.

For $0 < x < \epsilon$, we have: $\tilde{ReLU}(x) = 0 = ReLU(x) - ReLU(x)$, since x will not change in the execution, i.e. $ReLU(x) - \tilde{ReLU}(x) = x$. The error also scales linearly with the number of execution times.

In conclusion, the ReLU operator is either not affected by the error or linearly affected by the execution times.

Sigmoid. Let I denote the input $x \in I$, the *sigmoid* operator can be described as follows.

$$Sig = \frac{1}{1 + e^{-I}}$$

Similarly, let ϵ denote the absolute error on the input, where $\epsilon \ll I$, $\tilde{Sig}$ denote the actual value, then we have:

$$\begin{aligned} \tilde{Sig} &= Sig(x) + Sig^{'}\epsilon \\ &= Sig(x) + Sig(x)(1 - Sig(x))\epsilon \end{aligned} \tag{4}$$

We can conclude that the difference between the real value and the actual value grows linearly as the time of execution increases for the sigmoid operator.

Tanh. Let I denote the input, $x \in I$, the *tanh* operator can be described as follows.

$$T(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

Similarly, let ϵ denote the absolute error on the input, where $\epsilon \ll I$, $\tilde{T}$ denote the actual value, then we have:

$$\begin{aligned} \tilde{T}(x) &= T(x) + T^{'}\epsilon_x \\ &= T(x) + (1 - (T(x))^2)\epsilon \end{aligned} \tag{5}$$

Since $1 - (T(x))^2$ is a constant when x is given. Thus the *tanh* operator is linearly affected by the execution times.

Softmax. The description of the *softmax* operator is shown as follows. Similarly, let ϵ denote the absolute error on the input, where $\epsilon \ll I$, $\tilde{Soft}$ denote the actual value. Since deriving $Softmax$ is more complicated, we prefer to give the conclusion for it directly.

$$\tilde{Soft}(x) = Soft(x) + Soft^{'}\epsilon_x \tag{6}$$

where

$$Soft^{'}(x) = \begin{cases} Soft_{x_i}(1 - Soft_{x_j}), x_i \neq x_j \\ -Soft_{x_j}Soft_{x_i}, x_i = x_j \end{cases}$$

Since $Soft^{'}\epsilon_x$ is a constant when I is given, thus the *softmax* operator is linearly affected by the execution times. Following such practices, the precision error is accumulated, which helps explain the operator's precision impact on models. According to the error analysis, the error inside the convolution operator grows as n increases. For the pooling operator and other non-linear operators, the error scales linearly with the number of execution times. Moreover, the finite representation bit of floating-point numbers makes it more difficult to config an appropriate threshold.

3.5 Guiding Strategy

To improve test efficiency, heuristic algorithms are employed, and intermediate testing inputs are collected. We introduce three guiding strategies in Predoo, i.e., *random*, l_∞ *error*, l_1 *error*. Like random testing, the random strategy provides no guidance in the input generation, i.e., each input is independent of others. To employ heuristic-guided strategies, we introduce error-guided strategies. More specifically, error is calculated by l_{norm} distance. Formally, l_{norm} distance between tensor t and t' of the same shape can be described as follows.

$$l_{norm}(t, t') = (\Sigma_{i=1}^{d}(||t^i - t'^i||^k))^{(1/k)}$$

l_∞ *error* is measured by l_∞ distance, while l_1 *error* is measured by l_1 distance. The l_∞ *error* strategy focus on the maximized element-wise precision. Once a higher l_∞ is triggered, the corresponding input will be added back to the corpus. Mutation methods then help explore the surrounding space of the candidate input. Because l_∞ *error* focus on local errors inside the input, we propose l_1 *error* to implement the overall error estimation of the output.

3.6 Execution Evaluation

Similar to other numerical precision testing work, we measure the relative error produced by operator instances. We define two metrics to evaluate the current test input. If an input triggers an exception or a crash, it will be recorded as an operator defect. Corresponding inputs and the found precision error are saved for further analysis by library developers. However, in precision testing, we do not always run into such exceptions, i.e., an explicit oracle. We also want to catch smaller changes reflected by the output, i.e., precision errors, an implicit oracle.

Outputs produced by inputs and operators produced by testing preparation are used to approximate the implicit oracle. Similarly, we record element-wise precision to measure the errors. Predoo works to find the maximum precision errors during the execution. Therefore, the measured precision error will not be directly used to evaluate the quality of an operator.

4 EVALUATION

To evaluate the effectiveness and efficacy of the proposed approach, we implement our approach in a tool named Predoo[2]. Two baselines were selected in the evaluation.

1) **Built-in threshold**. The built-in threshold is the threshold TensorFlower declared in the source code when testing a DL operator. We compare the error found by Predoo with the build-in error threshold to evaluate the ability to find precision errors.
2) **CRADLE**[3], the state-of-the-art tool to locate error operators in DL libraries. Since the source code is not available, we implement our copy of it following the introduced technical approach. CRADLE is a state-of-the-art tool in testing DL libraries. It performs cross-implementation inconsistency checking and intermediate output analysis on DL models to detect defects in DL operators. We compare Predoo with CRADLE to evaluate the ability to locate operator defects.

4.1 Experimental Setup

We run the experiment on two workstations. One is a GNU/Linux System with Ubuntu 20.04. It is equipped with Intel Core CPU i7-6580K (6 cores, 3.6GHz), NVIDIA Corporation GP102 GPUs, and 64GB RAM. The CUDA version is 10.2, the Anaconda version is 5.2.0, and the Python version is 3.6. The other is a Windows System with Windows 10 2019. It is equipped with Intel Core CPU i7-9570H (6 cores, 2.6GHz), and 16GB RAM. The Anaconda version is 4.8.3, and the Python version is 3.6.

We select seven operators from TensorFlow in the experiment, including ***conv2d***, ***norm***, ***pooling***, ***relu***, ***sigmoid***, ***softmax***, and ***tanh***. We include operators from three released TensorFlow versions to avoid potential bias, i.e., TensorFlow 2.0.0, TensorFlow 2.3.1, TensorFlow 2.4.0. TensorFlow is chosen for evaluation considering its popularity. Also, operators we experiment with are widely used to construct DL models. They share few similarities in functionality and implementation.

Each operator is tested with 20, 000 inputs, i.e. $tn = 20,000$. We divide the test inputs into 20 groups according to chronological order to simplify the error representation record. We record the largest error of a group g as le_g, and the largest error $\varepsilon = max(le_1, ..., le_{20})$.

4.2 Research Questions

The goal of the experiment is to answer the following research questions:

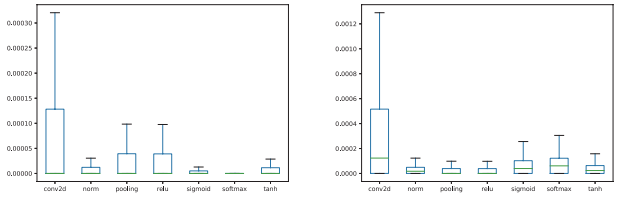
RQ1 How input precision and computation precision impact DL operators?
RQ2 How effective is Predoo at finding error-inducing inputs for operators?
RQ3 Which guiding strategy is more effective in generating higher error-inducing inputs?

As mentioned in the previous section, DL operators can process inputs of different precision with different precision. **RQ1** is designed to figure out how their impacts. To quantify the performance of Predoo, we design **RQ2**. It is used to evaluate the efficiency and efficacy of Predoo. Besides, three strategies are introduced with the hope of improving the efficiency of error-inducing input generation. We compare these strategies in **RQ3**.

[2]The source code of Predoo is available on Github, https://github.com/predoodl/predoo
[3]We implement our copy of CRADLE following the introduced technical approach because the source code is not found/not available, https://github.com/predoodl/cradle

4.3 Precision Impact

The input precision change is implemented by converting the input value to the data type with a different bit length. More specifically, we make use of the *convert_to_tensor* method provided in TensorFlow to implement typecast. Figure 2a shows the distribution of corresponding precision error caused by input precision changes. For simplicity, we do not plot outliers in Figure 2.



(a) input precision changes **(b) operator precision changes**

Figure 2: Precision impact

As we can conclude, *conv2d* is most sensitive to input precision changes, where the measured precision error $\varepsilon \in [0, 3.2e{-}4]$. *softmax* behaves best among all the observed operators, where the measured precision error $\varepsilon = 0$. The impacts of input precision changes on *pooling* and *relu* are similar, where the measured precision error $\varepsilon \in [0, 1.0e{-}4]$. Also, for *norm* and *tanh*, the measured precision error $\varepsilon \in [0, 3.5e{-}4]$.

To implement operator precision changes, we configure operator running with different data types. Figure 2b shows the corresponding precision error distribution caused by operator precision changes. Similarly, *conv2d* is the most sensitive to operator precision changes, where the measured precision error $\varepsilon \in [0, 1.2e{-}3]$. All operators are affected by operator precision changes. *relu* and *pooling* perform better than other DL operators, where the measured precision error $\varepsilon \in [0, 1.5e{-}4]$.

Besides, the maximum precision error triggered on each operator is recorded in Table 1. For the seven DL operators, operator precision changes cause a higher error, especially for the *conv2d* operator, where the triggered error is 3.56e−3. Considering the error analysis result for each DL operator, operators containing multiplication operands are more likely to trigger a larger precision error when the operator precision changes.

Answer to RQ1: Operator precision changes trigger larger precision errors in the output, comparing to input precision changes. *conv2d* is the most precision sensative operator, while *softmax* performs best with respect to input precision changes.

4.4 Effectiveness of Predoo

We estimate the effectiveness of Predoo from two perspectives, i.e., precision errors it triggers and operator defects it exposes.

Table 1: Largest error for each operator

	input precision changes	*operator precision change*
conv2d	2.84e-3	**3.56e-3**
norm	1.69e-4	**2.53e-4**
pooling	4.95e-4	4.95e-4
relu	7.84e-4	7.84e-4
sigmoid	1.22e-4	**3.99e-4**
softmax	2.28e-4	**3.66e-4**
tanh	1.52e-4	**2.77e-4**

Precision errors. An assumption we hold is that each DL operator under test should pass the test in the source repository, which means the test inputs used by TensorFlowers cannot trigger precision errors that exceed the threshold. Therefore, we use the built-in threshold, i.e., errors defined in the test code for each DL operator from TensorFlow, as a baseline. Meanwhile, we collect the maximum precision errors founded by Predoo. Figure 3 shows the result of the maximum precision errors found on each operator. Because there is no testing code available in the TensorFlow repository for the *sigmoid* and *tanh* operator, we mark the built-in threshold for them as N/A.

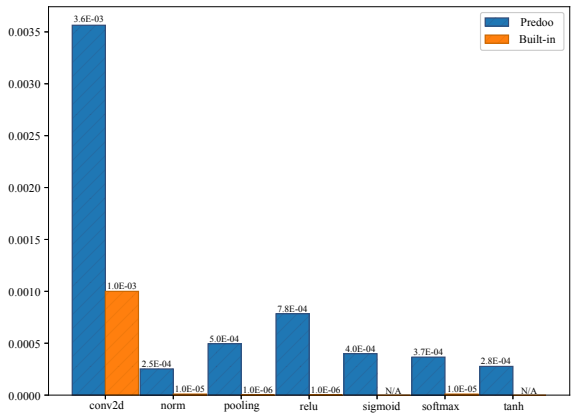


Figure 3: Testing results for DL operators

As can be concluded, compared to the built-in baseline, Predoo can trigger higher errors for all the DL operators under test. The error found by Predoo is at least three times larger than the built-in threshold. Meanwhile, we present a comparison between the test input scale, and we also record the execution time of Predoo in Table 2. As we run Predoo on operators from libraries of different versions on both CPU and GPU, we use an interval to record the time cost.

Predoo can produce many more test inputs to the DL operator than the test inputs available in the built-in test scripts. Meanwhile, Predoo takes no more than eight minutes to complete precision testing tasks. Compared to existing DL library testing approaches like CRADLE, Predoo outperforms these approaches in cost because no

Table 2: Comparison between Predoo and built-in test

	Test inputs		*Predoo execution time*
	Predoo	*Built-in*	*(seconds)*
conv2d	20000	32	[268, 371]
norm	20000	48	[276, 434]
pooling	20000	4	[96, 158]
relu	20000	1	[81, 113]
sigmoid	20000	N/A	[71, 105]
softmax	20000	1	[68, 106]
tanh	20000	N/A	[77, 107]

Table 3: Result of CRADLE

	MNIST+LeNet	VGG-16+Cifar-10
training time	10min	60-120min
CRADLE	10s	30s
operator	conv2d pooling ...	conv2d pooling norm ...
problem	N/A	N/A

extra time is required to perform model training and intermediate state extraction. Though a pre-trained neural network can save the training time, it could be difficult for uncommon DL operators to prepare such pre-trained models. In our experiment, we experiment CRADLE on two models, i.e., LeNet with MNIST and VGG-16 with CIFAR-10. Time cost for the whole process is shown in Table 3. LeNet and VGG-16 are constructed and tested in our experiment. For simplicity, we list operators also tested in Predoo in the table. Other operators are neither studied in our experiment nor reported buggy by CRADLE. As we can see, the extra time cost is expensive for CRADLE, especially when the model becomes complex, e.g., for VGG-16, more than one hour is required to finish the training task on GPU. Besides, CRADLE fails to expose precision errors for the *conv2d* operator in analyzing the intermediate output, while Preoo succeeds in finding larger precision errors than the built-in threshold provided by TensorFlowers.

Operator defects. Operator defects can lead to execution exceptions during the testing. They work as explicit oracles when testing DL operators, and such defects are considered more severe than precision error. Predoo can also detect operator defects in DL operators. In the experiment, we find and report two operator defects to the library developers. CRADLE cannot find these defects because DL models fail to call these APIs.

The first defect we come across is that we fail to run the *norm* operator in TensorFlow 2.0.0. The operator call throws the Valid device NotFoundError. While the *norm* operator in TensorFlow 2.3.1 and the *norm* operator in TensorFlow 2.4.0 do not throw any exceptions. According to the response from TensorFlowers, it is a known issue that is fixed in newly released versions.

The second defect we run into is that the *conv2d* operator fails to process tensor input in NCHW format. According to the description, *conv2d* should support both NHWC and NCHW format.

However, when we use a input that is transposed from NHWC format to NCHW format with *trans*, it throws UnimplementedError or InvalidArgumentError when value of *op_params* changes. It is confirmed by TensorFlowers to be an API-related bug. The problem remains in the latest release TensorFlow 2.4.1, and they are still working on the solution.

Answer to RQ2: Predoo can significantly expose precision errors inside DL operators, and the errors found are more than one order of magnitude larger on average. Predoo outperforms CRADLE in locating operator errors in time cost. Besides, Predoo can also expose API-related bugs when testing DL operators.

4.5 Strategy Comparison

To better evaluate the strategies we introduced in Predoo, we experiment on testing these DL operators. We use each of the introduced strategies to test operators from TensorFlow of different versions on both CPU and GPU. In this comparison, we treat *random* as the baseline as there is no guidance.

Figure 4 shows the result, l_∞ error and l_1 error performs better than *random* in triggering precision error for the *conv2d* and *relu* operator. However, for other operators, the performance of these strategies is similar.

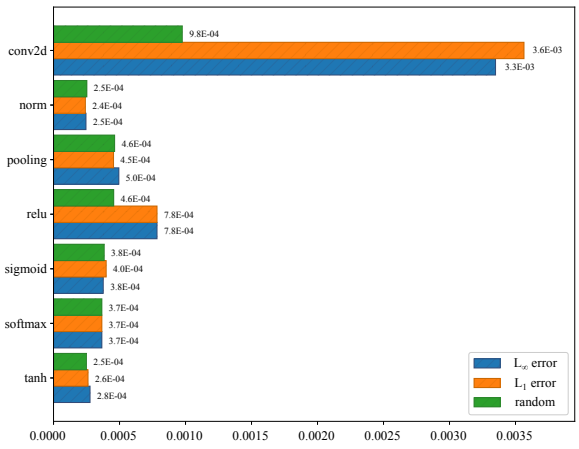


Figure 4: Testing results with different strategies

Therefore, we conduct a more detailed investigation on strategy impact on precision errors by analyzing the error distribution. Figure 5 shows the result. Similarly, outliers are not plotted in this figure. For all DL operators, *random* generate lower error-inducing inputs, while l_1 error performs a little better than l_∞ error.

After analyzing the data, we put forward a possible explanation. Because *random* is implemented with no guidance in test input generation, it is difficult to hit the larger error-inducing inputs. As a result, if a larger error can be triggered, l_∞ error and l_1 error perform better in guiding the exploration of the input space. Meanwhile, considering the Lipschitz continuity property hold by DL operators,

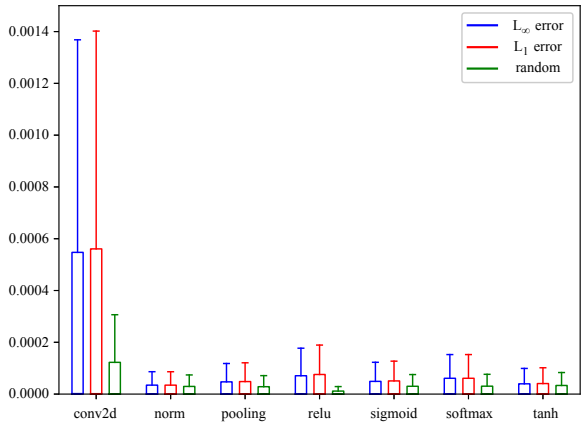


Figure 5: Error distribution with different strategies

l_∞ error and l_1 are more likely to hit larger error-inducing inputs. Although the error distribution of *random* is smaller than other strategies, it triggers competitive precision error because there is sufficient test data.

Answer to RQ3: l_∞ error and l_1 error are more effective than *random* in guiding the generation of larger error-inducing inputs. As *random* can achieve competitive performance in five out of the seven tested operators, we think *random* is also helpful for operators with smaller precision errors.

5 DISCUSSION

In this section, we discuss the practices of enterprise collaborators in Huawei and explain the usefulness and the limitation of Predoo. When locating causes of large precision error inside DL operators found by Predoo, library developers in Huawei found error-inducing practices. A typical example is that to achieve faster computing performance, they reduced the iteration times when applying Newton's method for approximation. Consequently, the implementation failed to converge to the root and produce large precision errors. They hold the opinion that industrial library developers widely adopt such practices in implementing and optimizing DL operators. Unlike existing numerical programs, DL operators are expected to run on various devices with different finite-bit representations. In addition to exceptions and crashes, precision errors are worth noting. In Predoo, we design and implement fine-grained precision error evaluation for DL operators. With the high precision error found by Predoo, library developers can review and optimize the operator's implementation. Besides, Predoo can also trigger API-related defects in operators.

Since Predoo focuses on exposing precision error inside DL operators, it also has limitations. Suppose Predoo reports no precision error. In that case, we cannot conclude that the DL operator is well implemented because it could also be that the operator's implementation fails to react to the changes in precision.

6 THREATS TO VALIDITY

In our experiment, we selected seven widely used DL operators in building DL models. Besides, according to the implementation of how these operators process inputs, they can be divided into different categories, i.e., convolution, pooling, and non-linear operators. All these operators are widely used in constructing DL models. Meanwhile, Differences in their functionality and implementation help ensure diversity. Therefore, we can prove the applicability of Predoo. Besides, Predoo is adopted by our enterprise collaborators. Predoo does not require any prior knowledge about the operator.

The experiment is conducted on two devices to avoid any inherent bias brought by the environment. When studying the impact of input precision and operator precision changes, we use error distribution instead of the maximum precision error to prove its soundness. Consider the type-variable characteristic of DL operators. We investigate how precision changes in the input and the operator impact the output. To avoid any haphazardness, we collect different operators from TensorFlow 2.0.0, TensorFlow 2.3.1, and TensorFlow 2.4.0. We compare the precision error found by Predoo against the built-in error threshold. We do not choose numerical precision testing approaches because these approaches can only process scala or small-size vectors. They cannot generate inputs in high dimensions. Besides, other existing work may employ symbolic execution to execute more paths in the operator. However, most DL operators do not have many branches inside their implementation. It is time-wasting to analyze its structure.

Although we do not select existing black-box testing approaches, we implement *random* strategy as random testing for DL operators instead as a baseline for black-box testing approaches. We evaluate the effectiveness of the proposed strategies in finding precision errors and prove that the l_∞ *error* strategy and the l_1 *error* strategy performs better than *random* strategy. Besides, we compare Predoo with the state-of-the-art DL library testing approach, CRADLE. The experimental result shows that Predoo can expose precision-related errors with less time required.

7 RELATED WORK

7.1 Numerical Testing

From the perspective of software testing, absolute error and relative error help check against the bound. Intuitively, if more exceptions or a larger error can be triggered and monitored during execution, the testing approach is considered more promising in exposing potential defects. To predict the stability of a floating-point program, Bao [4] detects possible places where an error becomes substantially inflated with an appropriate threshold. Barr [6] proposes a symbolic execution approach to detect floating-point exceptions. Bagnara [3] presents a symbolic evaluator for numerical program paths. Although these approaches could be effective, they deal with operands for the scalar. Guo [12] extends symbolic execution to test matrix in FPGen. However, it requires extra knowledge about the input specification. Chiang [8] treats input generation as a searching problem and implements a binary guided heuristic search algorithm to find larger error-inducing inputs, which focuses on operators that process primitive data types. To detect numerical bugs in DL models, Zhang [42] conducts static analysis for detecting numerical bugs at the architecture level.

Existing testing techniques could be difficult to apply to DL operators. Implementation of these operators usually does not have many branches, making it meaningless to explore paths. The high dimension of tensors adds to the complexity of space searching exponentially, making input range division not applicable.

7.2 DL Library Testing

DL library testing focuses on potential defects inside DL libraries, making it different from deep neural network(DNN) testing. Existing DNN testing techniques [29] [37] treat a DNN together with its underlying DL libraries as a whole, i.e., they cannot distinguish model defects from library defects. As a result, libraries' defects will be introduced to the trained models and impact their performance, which could further mislead model repairment.

Islam conducted a comprehensive study on DL bug characteristics and discussed the significant effects of bugs in [15]. Researchers begin to realize the importance of investigating DL libraries. Jia [17] reported that major reported bugs reside in DL algorithms and their interfaces, about 11.79% and 26.42%, respectively. Nejadgholi [26] conducts an empirical study to figure out how researchers and practitioners approximate the oracle when testing deep learning libraries. To identify and locate such defects, Pham [30] proposes a cross-backend validation approach to trace operators that cause inconsistent behaviors in the model output. With the help of high-level libraries like Keras [11], they can build models running on different DL libraries for output comparison and error location. Similarly, Guo [13] proposes Audee, which can also detect potential defects introduced in weights. Audee makes use of multiple DL libraries to implement consistency checks. However, we do not use Audee as a baseline because it checks library-irrelevant model defects in weights and parameters compared to CRADLE. Therefore, we select CRADLE, a library testing tool for comparison, although it is one year older. Also, Wang [35] proposes a series of mutation rules in LEMON to build DL models on different frameworks.

Different from existing DL library testing work, Predoo focuses on finding larger error-inducing inputs. Meanwhile, we do not require operators from different libraries because the same operator with a different precision can estimate the error.

8 CONCLUSION

In this paper, we propose Predoo, a precision testing approach to detect operator defects. To the best of our knowledge, Predoo is the first precision testing work for DL operators. Predoo implements a fuzzing-based approach is implemented to generate sufficient test inputs. Besides, expert knowledge is required to specify the acceptable error threshold. Predoo transfers the error-bound estimation problem to a searching problem to solve the problem, i.e., finding the maximum precision error triggered by test inputs. To evaluate its effectiveness, we conduct an experiment on seven different DL operators from TensorFlow.

Our result shows that Predoo can expose precision errors inside DL operators effectively. Also, Predoo can expose API-related defects in operators. The proposed guiding strategies generate error-inducing inputs, including l_∞ error-guided input generation and l_1 error-guided input generation strategy, which are more effective in generating higher error-inducing tensor inputs.

ACKNOWLEDGEMENTS

We want to thank anonymous reviewers for their insightful comments. This work is supported partially by National Natural Science Foundation of China (61932012, 61802171) and Fundamental Research Funds for the Central Universities (14380021).

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: A System for Large-scale Machine Learning. In *{USENIX} Symposium on Operating Systems Design and Implementation*. 265–283. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi

[2] Rachel Anderson, Hui Li, Yu Ji, Peifang Liu, and Maryellen L. Giger. 2019. Evaluating Deep Learning Techniques for Dynamic Contrast-enhanced MRI in The Diagnosis of Breast Cancer. In *Medical Imaging 2019: Computer-Aided Diagnosis, San Diego, California, United States, 16-21 February 2019 (SPIE Proceedings, Vol. 10950)*, Kensaku Mori and Horst K. Hahn (Eds.). SPIE, 1095006. https://doi.org/10.1117/12.2512667

[3] Roberto Bagnara, Matthieu Carlier, Roberta Gori, and Arnaud Gotlieb. 2013. Symbolic Path-oriented Test Data Generation for Floating-point Programs. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. IEEE, 1–10. https://doi.org/10.1109/ICST.2013.17

[4] Tao Bao and Xiangyu Zhang. 2013. On-the-fly Detection of Instability Problems in Floating-point Program Execution. In *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*. 817–832. https://doi.org/10.1145/2509136.2509526

[5] Earl T Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2014. The Oracle Problem in Software Testing: A Survey. *IEEE Transactions on Software Engineering* 41, 5 (2014), 507–525. https://doi.org/10.1109/TSE.2014.2372785

[6] Earl T Barr, Thanh Vo, Vu Le, and Zhendong Su. 2013. Automatic Detection of Floating-point Exceptions. *ACM Sigplan Notices* 48, 1 (2013), 549–560. https://doi.org/10.1145/2429069.2429133

[7] Yoshua Bengio, Yann LeCun, et al. 2007. Scaling Learning Algorithms Towards AI. *Large-scale Kernel Machines* 34, 5 (2007), 1–41.

[8] Wei-Fan Chiang, Ganesh Gopalakrishnan, Zvonimir Rakamaric, and Alexey Solovyev. 2014. Efficient Search for Inputs Causing High Floating-point Errors. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 43–52. https://doi.org/10.1145/2555243.2555265

[9] Anthony Di Franco, Hui Guo, and Cindy Rubio-González. 2017. A Comprehensive Study of Real-world Numerical Bug Characteristics. In *IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 509–519. https://doi.org/10.1109/ASE.2017.8115662

[10] Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. 2016. *Deep Learning*. Vol. 1. MIT Press Cambridge. https://doi.org/10.1007/s10710-017-9314-z

[11] Antonio Gulli and Sujit Pal. 2017. *Deep Learning with Keras*. Packt Publishing Ltd.

[12] Hui Guo and Cindy Rubio-González. 2020. Efficient generation of error-inducing floating-point inputs via symbolic execution. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, 1261–1272. https://doi.org/10.1145/3377811.3380359

[13] Qianyu Guo, Xiaofei Xie, Yi Li, Xiaoyu Zhang, Yang Liu, Xiaohong Li, and Chao Shen. 2020. Audee: Automated Testing for Deep Learning Frameworks. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 486–498. https://doi.org/10.1145/3324884.3416571

[14] Richard Hamlet and J Maciniak. 1994. Random Testing. Encyclopedia of Software Engineering. *Wiley: New York* (1994), 970–978.

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

[16] Anastasiia Izycheva and Eva Darulova. 2017. On Sound Relative Error Bounds for Floating-point Arithmetic. In *Formal Methods in Computer Aided Design*. IEEE, 15–22. https://doi.org/10.23919/FMCAD.2017.8102236

[17] 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 (Lecture Notes in Computer Science, Vol. 12112)*, Yunmook Nah, Bin Cui, Sang-Won Lee, Jeffrey Xu Yu, Yang-Sae Moon, and Steven Euijong Whang (Eds.). Springer, 604–620. https://doi.org/10.1007/978-3-030-59410-7_40

[18] Xiaotang Jiang, Huan Wang, Yiliu Chen, Ziqi Wu, Lichuan Wang, Bin Zou, Yafeng Yang, Zongyang Cui, Yu Cai, Tianhang Yu, Chengfei Lv, and Zhihua Wu. 2020. MNN: A Universal and Efficient Inference Engine. In *MLSys*. https://proceedings.mlsys.org/book/287.pdf

[19] Tamara G Kolda and Brett W Bader. 2009. Tensor Decompositions and Applications. *SIAM review* 51, 3 (2009), 455–500. https://doi.org/10.1137/07070111X

[20] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems*. 1097–1105. https://proceedings.neurips.cc/paper/2012/hash/c399862d3b9d6b76c8436e924a68c45b-Abstract.html

[21] Ignacio Laguna. 2019. FPChecker: Detecting Floating-point Exceptions in GPU Applications. In *IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 1126–1129. https://doi.org/10.1109/ASE.2019.00118

[22] Marius Leordeanu. 2020. *Unsupervised Learning in Space and Time - A Modern Approach for Computer Vision using Graph-based Techniques and Deep Neural Networks*. Springer. https://doi.org/10.1007/978-3-030-42128-1

[23] Weisi Luo, Dong Chai, Xiaoyue Run, Jiang Wang, Chunrong Fang, and Zhenyu Chen. 2021. Graph-based Fuzz Testing for Deep Learning Inference Engines. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 288–299. https://doi.org/10.1109/ICSE43902.2021.00037

[24] William M McKeeman. 1998. Differential Testing for Software. *Digital Technical Journal* 10, 1 (1998), 100–107. http://www.hpl.hp.com/hpjournal/dtj/vol10num1/vol10num1art9.pdf

[25] Charlie Miller, Zachary NJ Peterson, et al. 2007. Analysis of mutation and generation-based fuzzing. *Independent Security Evaluators, Tech. Rep* 56 (2007), 127–135.

[26] Mahdi Nejadgholi and Jinqiu Yang. 2019. A Study of Oracle Approximations in Testing Deep Learning Libraries. In *IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 785–796. https://doi.org/10.1109/ASE.2019.00078

[27] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An Imperative Style, High-performance Deep Learning Library. In *Advances in Neural Information Processing Systems*. 8026–8037. https://proceedings.neurips.cc/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html

[28] Vern I Paulsen and RR Smith. 1987. Multilinear Maps and Tensor Norms on Operator Systems. *Journal of functional analysis* 73, 2 (1987), 258–276. https://doi.org/10.1016/0022-1236(87)90068-1

[29] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. 2017. Deepxplore: Automated Whitebox Testing of Deep Learning Systems. In *Symposium on Operating Systems Principles*. 1–18. https://doi.org/10.1145/3132747.3132785

[30] 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 *International Conference on Software Engineering*. IEEE, 1027–1038. https://doi.org/10.1109/ICSE.2019.00107

[31] Wenjie Ruan, Xiaowei Huang, and Marta Kwiatkowska. 2018. Reachability Analysis of Deep Neural Networks with Provable Guarantees. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden*, Jérôme Lang (Ed.). ijcai.org, 2651–2659. https://doi.org/10.24963/ijcai.2018/368

[32] Stephen F Siegel, Anastasia Mironova, George S Avrunin, and Lori A Clarke. 2008. Combining Symbolic Execution with Model Checking to Verify Parallel Numerical Programs. *ACM Transactions on Software Engineering and Methodology* 17, 2 (2008), 1–34. https://doi.org/10.1145/1348250.1348256

[33] Laura Titolo, Marco A Feliú, Mariano Moscato, and César A Muñoz. 2018. An Abstract Interpretation Framework for The Round-off Error Analysis of Floating-point Programs. In *International Conference on Verification, Model Checking, and Abstract Interpretation*. Springer, 516–537. https://doi.org/10.1007/978-3-319-73721-8_24

[34] Petar Tsankov, Mohammad Torabi Dashti, and David Basin. 2013. Semi-valid Input Coverage for Fuzz Testing. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*. 56–66. https://doi.org/10.1145/2483760.2483787

[35] Zan Wang, Ming Yan, Junjie Chen, Shuang Liu, and Dongdi Zhang. 2020. Deep Learning Library Testing via Effective Model Generation. In *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 788–799. https://doi.org/10.1145/3368089.3409761

[36] W Eric Wong, Joseph R Horgan, Saul London, and Hiralal Agrawal. 1997. A Study of Effective Regression Testing in Practice. In *International Symposium on Software Reliability Engineering*. IEEE, 264–274. https://doi.org/10.1109/ISSRE.1997.630875

[37] 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 *ACM SIGSOFT International Symposium on Software Testing and Analysis*. 146–157. https://doi.org/10.1145/3293882.3330579

[38] Bing Xu, Naiyan Wang, Tianqi Chen, and Mu Li. 2015. Empirical Evaluation of Rectified Activations in Convolutional Network. *arXiv preprint arXiv:1505.00853* (2015). http://arxiv.org/abs/1505.00853

[39] Dong Yu, Adam Eversole, Mike Seltzer, Kaisheng Yao, Zhiheng Huang, Brian Guenter, Oleksii Kuchaiev, Yu Zhang, Frank Seide, Huaming Wang, et al. 2014. *An Introduction to Computational Networks and The Computational Network Toolkit*.

Technical Report. Microsoft Technical Report MSR-TR-2014–112.

[40] Xiaoyong Yuan, Pan He, Qile Zhu, and Xiaolin Li. 2019. Adversarial Examples: Attacks and Defenses for Deep Learning. *IEEE Transactions on Neural Networks and Learning Systems* 30, 9 (2019), 2805–2824. https://doi.org/10.1109/TNNLS.2018.2886017

[41] Yuhao Zhang, Yifan Chen, Shing-Chi Cheung, Yingfei Xiong, and Lu Zhang. 2018. An Empirical Study on TensorFlow Program Bugs. In *ACM SIGSOFT International Symposium on Software Testing and Analysis*. 129–140. https://doi.org/10.1145/3213846.3213866

[42] Yuhao Zhang, Luyao Ren, Liqian Chen, Yingfei Xiong, Shing-Chi Cheung, and Tao Xie. 2020. Detecting Numerical Bugs in Neural Network Architectures. In *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 826–837. https://doi.org/10.1145/3368089.3409720

[43] Peiyuan Zong, Tao Lv, Dawei Wang, Zizhuang Deng, Ruigang Liang, and Kai Chen. 2020. FuzzGuard: Filtering out Unreachable Inputs in Directed Grey-box Fuzzing through Deep Learning. In *USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, Srdjan Capkun and Franziska Roesner (Eds.). USENIX Association, 2255–2269. https://www.usenix.org/conference/usenixsecurity20/presentation/zong

[44] Daming Zou, Ran Wang, Yingfei Xiong, Lu Zhang, Zhendong Su, and Hong Mei. 2015. A Genetic Algorithm for Detecting Significant Floating-point Inaccuracies. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 529–539. https://doi.org/10.1109/ICSE.2015.70