When AllClose Fails: Round-Off Error Estimation for Deep Learning Programs

Qi Zhan*, Xing Hu*‡ Yuanyi Lin†, Tongtong Xu†, Xin Xia*, Shanping Li*
*The State Key Laboratory of Blockchain and Data Security, Zhejiang University, Hangzhou, China
†Huawei, Hangzhou, China
*{qizhan, xinghu, shan}@zju.edu.cn, xin.xia@acm.org
†{linyuanyi2, xutongtong9}@huawei.com

***Abstract*—Deep learning programs are continually enhanced for improved performance through the use of kernel-level optimizations, parallel training, and low-precision arithmetic. These optimizations provide different implementations that are mathematically equivalent. Round-off error in floating-point computations can lead to differences in the outputs of these implementations, even when the mathematical equivalence holds. When the outputs of customized and reference implementations exceed the tolerance thresholds, it is difficult for developers to distinguish between acceptable round-off errors and implementation bugs. This paper proposes an approach called RENDER to classify the numerical errors between two implementations based on estimating the maximum round-off error. RENDER combines dynamic interval arithmetic and round-off error analysis to compute scalable and tight output bounds. We demonstrate the effectiveness of our method on real-world issues by comparing it with the state-of-the-art tool, SATIRE and a High-Precision Re-execution baseline. Experimental results show that our approach identifies at least 25% more errors and achieves an average speedup of 19× compared to SATIRE, enabling developers to debug and optimize implementations more efficiently.**

***Index Terms*—Round-off error analysis, Deep learning programs, Interval arithmetic**

I. INTRODUCTION

Deep learning (DL) has become a cornerstone of modern computing, powering various applications, including computer vision [1], natural language processing [2], and code generation [3]. The rapid growth of DL has also brought a new need to optimize the performance and efficiency of DL programs to handle increasingly large models.

To accelerate training and inference in deep learning, developers propose various methods. (1) Kernel and neural network-level optimizations [4], [5]. Developers continuously improve algorithms such as tiling and kernel fusion to fully utilize GPU resources. (2) Distributed and parallel training [6]. To accommodate larger models, developers employ distributed training methods such as data parallelism [7], model parallelism [8] to scale training on multiple GPUs or machines. (3) Trade precision for performance by employing low-precision arithmetic such as FP16 [9] or even FP8 [10]. Conceptually, these approaches implement a mathematically equivalent program with different optimizations.

Since these multiple implementations are equivalent in mathematics, comparing their outputs to validate correctness is common practice. However, exact output matches are typically impossible in deep learning programs because floating-point computations are inherently imprecise due to finite-precision representation and round-off errors. These errors can accumulate and propagate through computations, potentially resulting in significant differences in the final outputs. Compared with traditional numerical computing, low-precision floating-point arithmetic is widely used in deep learning programs, which amplifies the impact of rounding errors further.

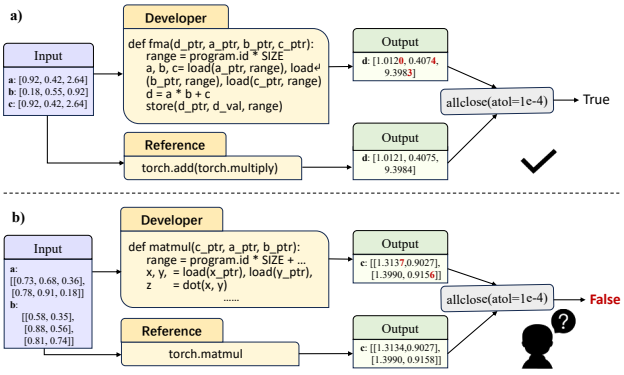


Fig. 1: Motivation of the "allclose" problem when comparing two implementations with a given input.

To address this, developers usually examine the outputs by absolute tolerances (atol) and relative tolerances (rtol). For instance, PyTorch provides a `torch.allclose` function, which determines whether two tensors are approximately equal by ensuring each pair of elements satisfies the condition: $|\text{input} - \text{reference}| \leq \text{atol} + |\text{rtol} \times \text{reference}|$. The specific values of atol and rtol are usually determined empirically or based on the datatype of the tensors.

As shown in Fig. 1a), the developer compares the output from fused multiply-add kernel with the reference implementation in PyTorch and believes the kernel is correct when `allclose` passes. The problem arises when `allclose` fails, as shown in Fig. 1b). The developer writes customized matrix multiplication and compares it with PyTorch's `matmul`, and it turns out that the two implementations produce different results that exceed the given tolerance. It is challenging to identify the root cause in this case.

‡Corresponding Author

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00016

The major concern of developers is whether they should refine the implementation further or safely ignore the error and continue developing. Therefore, the error and corresponding resolution can be classified into two types:

1) **Type-I: Acceptable Round-off error**, which is inevitable in floating-point computations. The error can be mitigated by tuning the numerical precision of certain variables. After balancing the trade-off between performance and precision, the developer can increase the numerical precision or adjust the atol and rtol to make the `allclose` pass.
2) **Type-II: Implementation bug**, which should be fixed by developers. This error may arise from incorrect algorithms, numerically unstable implementations, or faulty compiler optimizations. Developers should refine the implementation until the mismatch is resolved.

Compared to crashes or obvious results mismatches, debugging and analyzing the root cause of floating-point programs is notoriously challenging due to round-off errors. Developers must analyze the errors carefully and compare the intermediate step-by-step results, which is time-consuming and error-prone. The two implementations may adopt fundamentally different DL frameworks and algorithms, making it difficult to compare the intermediate results. Furthermore, the source code of the reference implementation is sometimes unavailable, as in the case of libraries such as cuBLAS, making it impossible to obtain the intermediate results.

The difficulty in identifying the root cause of errors can cause developer confusion and may even obscure real bugs. When facing output differences of a customized matrix multiplication kernel `tl.dot()` in Triton [11] with `matmul` in PyTorch [12], the developer remarked in the issue[1]:

> "Are there some bugs in tl.dot() or my test code? This precision error shouldn't be unacceptable."

Another example is a long-standing implementation bug in the gradient accumulation computation of Hugging Face Transformers training [13]. Gradient accumulation is commonly used to simulate larger batch sizes when memory constraints prevent training with large batches. A developer found that the training loss varied with batch size, which was initially misattributed to round-off error in 2021. The problem was eventually identified as an implementation error in gradient accumulation, which had persisted for three years before being fixed in 2024 [14]. This case illustrates that such implementation bugs can be subtle to diagnose and wrongly blamed on round-off errors.

The above examples illustrate the importance of classifying the two types of errors for developers. We formulate the problem as follows:

> **Problem:** Given a reference program and a target program, the output by the two programs differs for a given input. How can we determine whether the error is Type-I (round-off error) or Type-II (bug)?

[1]https://github.com/triton-lang/triton/issues/2843

Our Solution. To solve this problem, we propose a novel approach called RENDER, referred to **R**ound-Off **E**rror Estimatio**N** for **D**eep L**E**arning Prog**R**ams. RENDER estimates a round-off error bound rigorously for the target program and classifies the error into two types. The idea is illustrated in Fig. 2. Given a target program and the output (black dot), our approach estimates both the lower and upper bounds of the output. If the reference output falls within the bounds, we consider the error to be due to round-off error and classify it as Type-I (green dot). Developers can safely ignore the differences and adjust atol and rtol. Otherwise, we conclude that the error is due to implementation or compiler bugs and classify it as Type-II (red dot). Developers can further refine the implementations.

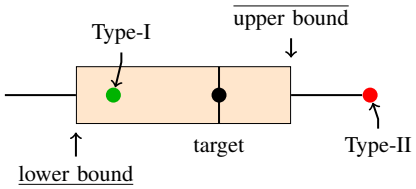


Fig. 2: The proposed method estimates the error bound for the target program.

Challenges: Compared to traditional programs, new challenges arise when traditional round-off error estimation approaches are applied to DL programs directly. (1) The computation involves a large number of operations, which requires a scalable and efficient approach to estimate the error. (2) The low-level implementation of operations is not fully transparent, such as the matrix multiplication details, making it difficult to estimate the bound accurately. To address these challenges, RENDER estimates the maximum floating-point round-off error dynamically based on the following key ideas:

❶ **Combine Interval Arithmetic and Error Analysis:** As computations in deep learning programs are typically large-scale, we adopt a fast interval arithmetic-based approach to estimate the interval of the output tensor for each operation. To estimate the round-off error for non-transparent operations, we combine the error analysis with the lower and upper bounds, which ensures the error bound is sound regardless of the low-level implementation.

❷ **Dynamic Analysis:** While conventional error analysis is typically static, the concrete input in our problems offers an opportunity to estimate errors dynamically. We rerun the target program and estimate the interval of each tensor during its execution. This allows us to avoid unnecessary overestimation for operations such as max, min, and conditional statements.

❸ **From Neural Network to Kernel-Level Bounds**: A DL program is typically a neural network composed of multiple kernels. Our approach focuses on kernel-level estimation first and automatically reduces the problem of the whole neural network to a series of kernel estimations, which simplifies the overall analysis and makes it scalable to large and complex deep learning programs.

We implement RENDER based on Triton [11] and PyTorch [12]. To evaluate our approach, we collect a dataset from real-world issues about precision errors in the deep learning community. There are 20 kernel-level test cases in total, 16 of which are Type-I and 4 of which are Type-II. We consider SATIRE [15], a state-of-the-art approach for computing rigorous rounding error bounds in large-scale programs and a High-Precision Re-execution baselines. Compared to baselines, our approach can successfully classify errors in 8 and 5 more cases, respectively. Among the successful cases, our approach always provides a tighter error bound compared with SATIRE. Furthermore, we evaluated our method on 7 real-world issues in neural networks and successfully analyzed 6 cases. Following the analysis, we submitted 5 feedback reports to the communities, with 2 already acknowledged by the developers. These results demonstrate the practical utility of our approach in addressing precision-related issues.

Contributions: Our contributions are as follows:

- To the best of our knowledge, this is the first work to classify the type of numerical error in the context of DL programs and provide a solution to the problem by estimating the maximum round-off error.
- We propose an approach to estimate the floating-point round-off error for a single kernel and show how to reduce the whole deep learning program to kernel levels.
- We evaluate our approach on a collection of DL programs derived from real-world issues and demonstrate its effectiveness and efficiency.

The rest of the paper is organized as follows. Section II presents the necessary background about floating-point numbers and error analysis. Section III describes our approach in detail. Section IV presents the experimental setup and evaluation. Section V provides a case study and discusses the results and limitations of our approach. Section VI discusses related work. Finally, Section VII concludes the paper.

II. PRELIMINARIES

This section presents the necessary background on floating-point representations, error models, and how to use interval arithmetic to estimate the error.

A. Floating-point Representation

Floating-point numbers approximate real numbers and can represent only a finite subset of the continuous real number space. According to the IEEE 754 standard [16], a floating-point number is composed of three components: the sign (s), the exponent (e), and the significand (m), and the value of the number is given by:

$$(-1)^s \times 2^{e-\text{bias}} \times 1.m,$$

where the bias is a constant that depends on the floating-point format. Table I illustrates the bit allocation of these components in half-, single-, and double-precision floating-point formats. In addition to the common types mentioned above, there are also many low-precision floating-point formats, such as bfloat16 (BF16) [17], TensorFloat32 (TF32) [18], FP8 [10], which are used widely in deep learning.

TABLE I: IEEE 754 Floating-Point Formats.

Format	Sign	Exponent	Significand	Machine Epsilon
Half	1	5	10	9.77×10^{-4}
Single	1	8	23	1.19×10^{-7}
Double	1	11	52	2.22×10^{-16}

The finite number of bits used to represent real numbers inevitably introduces inaccuracies. The nearest representable value may differ from the exact mathematical result by up to half a unit in the last place (ULP), and this difference is known as round-off error. IEEE 754 defines several rounding modes, and we mainly focus on the downward rounding mode, which rounds the result toward negative infinity, and the upward rounding mode, which rounds toward positive infinity. These modes ensure that the computed value is always less than or equal (in downward rounding) or greater than or equal (in upward rounding) to the exact result.

B. Error Model

As round-off error is inherently inaccurate in floating-point computations, it is essential to measure this error. Let $\text{fl}(\cdot)$ represent the result of an operation performed in floating-point arithmetic. The relative error can be expressed as:

$$Err_{rel} = \left| \frac{\text{fl}(x) - x}{x} \right|,$$

which quantifies how far the floating-point result is from the exact value. According to the standard error model in the textbook [19], we assume the following:

$$\text{fl}(x \text{ op } y) = (x \text{ op } y)(1+\delta), \delta \leq |u|,$$

where machine epsilon u is the upper bound on the relative error [20]. The machine epsilon values for different floating-point formats are provided in Table I.

This error model not only helps estimate the error of a single operation but also enables us to understand how errors propagate in multi-step computations. For example, consider a summation of n floating-point numbers $x_1, x_2, \ldots, x_n$, each with an error bounded by u. The error bound for the sum of these numbers, as discussed in [19], can be expressed as:

$$\Delta \leq (n-1)u \sum_{i=1}^{n} |x_i| + O(u^2). \tag{1}$$

This equation reveals that the relative error of the summation is primarily bounded by $(n-1)u$ times the sum of the absolute values of the input numbers. It is also worth noting that this error bound holds regardless of the order in which the summation is performed, which is particularly useful for DL programs where the summation order is not fixed.

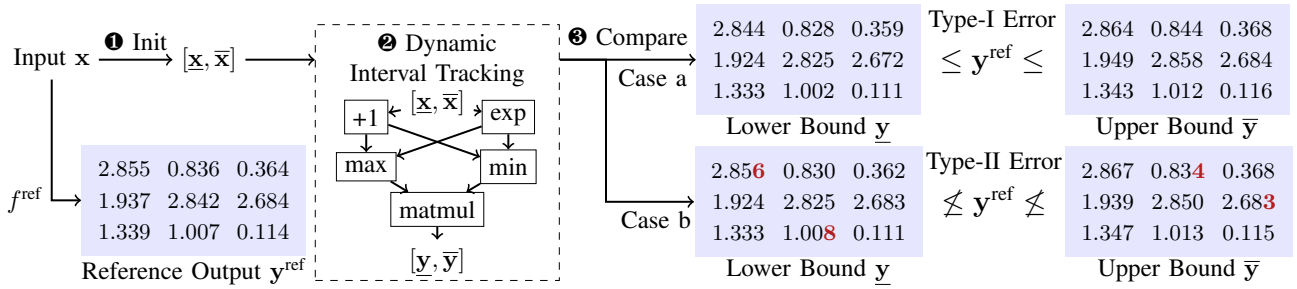


Fig. 3: Overview of RENDER. The highlighted number indicates the item that causes the reference to be out of the interval.

C. Interval Arithmetic

Interval arithmetic offers a rigorous approach for measuring the error in floating-point computations [21]. An interval represents a set of real values, which can be defined as: $[\underline{a}, \overline{a}] = \{a \in \mathbb{R} \mid \underline{a} \leq a \leq \overline{a}\}$, where $\underline{a}, \overline{a}$ are the lower and upper bounds, respectively [22]. For function f, interval arithmetic defines a corresponding function $\hat{f}$ between intervals, which ensures that $f(a_1, \ldots, a_n) \in \hat{f}([\underline{a_1}, \overline{a_1}], \ldots, [\underline{a_n}, \overline{a_n}])$, i.e., the result conservatively bounds all possible values arising from the corresponding real-number operations. For example, given two intervals $[a_1, a_2]$ and $[b_1, b_2]$, their addition and multiplication are defined as follows:

$$[\underline{a}, \overline{a}] + [\underline{b}, \overline{b}] = [\underline{a} + \underline{b}, \overline{a} + \overline{b}],$$
$$[\underline{a}, \overline{a}] \times [\underline{b}, \overline{b}] = [\min(\underline{ab}, \underline{a}\overline{b}, \overline{a}\underline{b}, \overline{ab}), \max(\underline{ab}, \underline{a}\overline{b}, \overline{a}\underline{b}, \overline{ab})]$$

Other operations can be defined similarly. Based on interval arithmetic, we can rigorously bound the error in floating-point computations by composing these elementary operations.

III. APPROACH

A. Overview

We illustrate how RENDER estimates the round-off error for a single kernel and extend it to the whole neural network later. The illustration is shown in Fig. 3. Given a target kernel f, a reference kernel f^{ref}, and an input tensor $\mathbf{x}$[2], let $\mathbf{y} = f(\mathbf{x})$ and $\mathbf{y}^{\text{ref}} = f^{\text{ref}}(\mathbf{x})$. When a divergence occurs between $\mathbf{y}$ and $\mathbf{y}^{\text{ref}}$, our goal is to estimate the possible interval of $\mathbf{y}$ produced by the target kernel f and determine whether the discrepancy corresponds to a Type-I or Type-II error.

❶ We begin by initializing the input interval as $[\underline{\mathbf{x}}, \overline{\mathbf{x}}] = [\mathbf{x}, \mathbf{x}]$, as the input tensor for two kernels are same. ❷ During the execution of the target kernel, we track the interval of each intermediate tensor. Each operation in the kernel updates its output interval based on the intervals of its inputs, following the standard rules of interval arithmetic. At the end of the computation, we obtain the output interval $[\underline{\mathbf{y}}, \overline{\mathbf{y}}]$. ❸ Compared to the reference output, there are two cases: (a) If the reference output $\mathbf{y}^{\text{ref}}$ falls within the interval, i.e. $\mathbf{y}^{\text{ref}} \in [\underline{\mathbf{y}}, \overline{\mathbf{y}}]$, meaning that each element y^{ref} satisfies $\underline{y} \leq y^{\text{ref}} \leq \overline{y}$, the error is attributed to round-off and classified as **Type-I**. (b) Otherwise, the error is classified as **Type-II**, indicating that it arise from implementation mismatches. This classification helps distinguish between acceptable round-off errors and potentially critical errors caused by bugs.

[2]We use **boldface** to denote tensors and italics to denote scalars.

B. Kernel Error Estimation

In this section, we detail the interval tracking step. We describe how to estimate the interval of the output tensor for each type of operation and how these intervals are propagated through the computation.

1) Arithmetic Error Estimation: Arithmetic operations are the primary source of round-off error. For each arithmetic operation, we compute the error interval using standard interval arithmetic. We have already discussed the interval arithmetic for addition and multiplication in Section II. For monotonic functions f such as the square root, we can apply them directly to the interval bounds, i.e. $\hat{f}([\underline{a}, \overline{a}]) = [f(\underline{a}), f(\overline{a})]$. It is worth noting that non-differentiable can also be handled, as long as if their semantics can be modeled using interval arithmetic. In addition to rounding errors, it is also necessary to account for ULP error, which arises in many operations, especially in hardware-accelerated computations and certain functions. For example, CUDA documentation [23] specifies that fast divide operations can introduce up to 2 ULP errors, and operations like exp can introduce 1 ULP error. We list ULP errors of common operations in Table II. By considering both rounding and ULP errors, we can provide a more sound estimate of the error introduced by each operation in the computation. Specifically, given an interval $[\underline{v}, \overline{v}]$ with machine epsilon ε and an operation-specific error parameter δ (measured in ULPs), we compute the new interval $[\underline{v'}, \overline{v'}]$ by:

$$\overline{v'} = \begin{cases} \overline{v}(1 + \varepsilon\delta), & \overline{v} \geq 0, \\ \overline{v}(1 - \varepsilon\delta), & \overline{v} < 0, \end{cases} \qquad \underline{v'} = \begin{cases} \underline{v}(1 - \varepsilon\delta), & \underline{v} \geq 0, \\ \underline{v}(1 + \varepsilon\delta), & \underline{v} < 0. \end{cases}$$

TABLE II: ULP error of common operations.

Operation	ULP Error
Addition / Subtraction	≤ 1
Multiplication	≤ 1
Fast Division	≤ 2
Square Root (sqrt)	≤ 1
Exponential (exp)	≤ 1

Algorithm 1: Matrix Multiplication Error Estimation

Input: $A^{[l,u]} \in \mathbb{R}^{n\times m}$, $B^{[l,u]} \in \mathbb{R}^{m\times k}$, multiplicative error ε_*, additive error ε_+
Output: $C^{[l,u]} \in \mathbb{R}^{n\times k}$: Interval bounds of AB

1 **for** $i \leftarrow 1$ **to** n **do**
2 **for** $j \leftarrow 1$ **to** k **do**
3 $a_i^{[l,u]}, b_j^{[l,u]} \leftarrow A^{[l,u]}[i,:], B^{[l,u]}[:,j]$;
4 $ll, lu, ul, uu \leftarrow a_i^l b_j^l,\ a_i^l b_j^u,\ a_i^u b_j^l,\ a_i^u b_j^u$;
5 $lo \leftarrow \min(ll, lu, ul, uu)$;
6 $hi \leftarrow \max(ll, lu, ul, uu)$;
7 $lo \leftarrow lo \cdot (1-\varepsilon_*),\ hi \leftarrow hi \cdot (1+\varepsilon_*)$;
8 $c^l \leftarrow \sum lo,\ c^u \leftarrow \sum hi$;
9 $c^l \leftarrow c^l - \sum |lo| \cdot (m-1) \cdot \varepsilon_+$;
10 $c^u \leftarrow c^u + \sum |hi| \cdot (m-1) \cdot \varepsilon_+$;
11 $C^{[l,u]}[i,j] \leftarrow [c^l, c^u]$;
12 **end**
13 **end**
14 **return** $C^{[l,u]}$

2) Precision Casting: It is common to cast the precision of one tensor to either higher or lower precision to optimize performance and improve memory efficiency [24]. The way we handle precision casting depends on whether we are casting to higher or lower precision:

- Higher Precision: When casting to a higher precision, the interval of the output tensor is directly obtained from the interval of the input tensor.
- Lower Precision: When casting to a lower precision, the interval of the output tensor may be affected by precision loss during the casting. We apply downward rounding for the lower bound and upward rounding for the upper bound, using the machine epsilon of the lower precision.

3) Non-Arithmetic Operations: Non-arithmetic operations such as `reshape`, `transpose`, and `gather` do not introduce numerical errors, as they do not involve any mathematical computations that could result in rounding or approximation. These operations typically only change the arrangement or structure of the data in the tensor, and the interval of the output tensor is simply propagated from the input tensor, with appropriate shape transformations.

4) Conditional Statement: Handling conditional statements is challenging in static error analysis [25], as the results depend on the specific branch executed, and overestimation of the error can occur when multiple branches are considered. As a dynamic approach, RENDER naturally supports control flow, since it observes the concrete execution path during runtime. For instance, in the case of the $\max$ operation, we assume two input a, b with corresponding intervals $[\underline{a}, \overline{a}]$ and $[\underline{b}, \overline{b}]$ respectively. The output $c = \max(a, b)$ will have an interval determinded by if $(a > b)$ then $[\underline{a}, \overline{a}]$ else $[\underline{b}, \overline{b}]$.

5) Summation and Matrix Multiplication: Matrix multiplication is the most important operation in deep learning, it should be handled carefully. Consider the matrix multiplication as $\mathbf{C} = \mathbf{AB}$, where $\mathbf{A} \in \mathbb{R}^{n\times m}, \mathbf{B} \in \mathbb{R}^{m\times k}, \mathbf{C} \in \mathbb{R}^{n\times k}$. The element-wise computation of the output tensor $\mathbf{C}$ can be expressed as:

$$C[i,j] = \sum_{q=1}^{m} A[i,q] \times B[q,j].$$

The computation of each element involves m multiplications and $m-1$ additions. The total error in matrix multiplication can be decomposed into three sources: (1) rounding error in multiplication, (2) accumulation error in summation, and (3) precision casting errors in hardware-specific implementations, such as TensorCore with TF32. The details of matrix multiplication in GPUs are tricky [26], as the precision and concrete order of addition are not fully clear.

We propose an error model that explicitly accounts for all three sources by incorporating multiplicative and additive error bounds. The algorithm is shown in Algorithm 1. We assume the elements of the input tensors are always positive to simplify the algorithm. We use ε_* to represent the machine epsilon used in the multiplication and ε_+ to represent the machine epsilon used in the summation. The whole process is a double loop to compute the output tensor element-wise. For each element $C[i,j]$, we first obtain the corresponding row of $\mathbf{A}$ and column of $\mathbf{B}$ (Line 3). Based on interval arithmetic of multiplication, we compute the lower and upper bounds of the product (Line 4-6). As the rounding error in the multiplication is less than 1 ULP, we round the multiplication result downward and upward, respectively (Line 7). In the end, we consider the summation error by Equation (1) and update the interval of the output tensor (Line 9-11). In addition to matrix multiplication, the summation operation can be considered a specific case of matrix multiplication by setting $m = 1$ in the algorithm. In this case, the error is only introduced by the summation operation.

C. Neural Network Error Classification

1) General Idea: In the previous section, we discussed error estimation for a single kernel. Now, we extend the analysis to neural networks composed of multiple kernels. Given a target program P consisting of layers $f_n \circ f_{n-1} \circ \cdots \circ f_1$, a reference program P^{ref} consisting of layers $f_n^{\text{ref}} \circ f_{n-1}^{\text{ref}} \circ \cdots \circ f_1^{\text{ref}}$, and an input tensor $\mathbf{x}$, we observe mismatches between the output $P(\mathbf{x})$ and $P^{\text{ref}}(\mathbf{x})$. Our objective remains the same: determining the root cause of the mismatches.

Since a neural network consists of multiple layers, we can naturally reduce the problem of mismatches in deep learning programs to a series of kernel-level problems. Starting from the first layer f_1 and f_1^{ref}, we apply the kernel-level estimation for this layer, and continue this process layer by layer until the last layer f_n and f_n^{ref}. The error is decided as Type-II if one layer is decided as Type-II. The main difference between single kernel and neural network error estimation is that we need to account for the different input tensors produced by the previous layers. As a result, the lower and upper bounds of

the initial intervals are not simply the same as the input value. For the i^{th} layer when $i > 1$, the initial interval is defined as

$$[\min(f_{i-1} \circ \cdots \circ f_1(\mathbf{x}), f^{\text{ref}}_{i-1} \circ \cdots \circ f^{\text{ref}}_1(\mathbf{x})),$$
$$\max(f_{i-1} \circ \cdots \circ f_1(\mathbf{x}), f^{\text{ref}}_{i-1} \circ \cdots \circ f^{\text{ref}}_1(\mathbf{x}))].$$

Here, $\min$ and $\max$ are applied element-wise, ensuring the computed bounds are large enough to enclose the true results.

2) Layer Approximation: In real-world deep learning programs, some layers are difficult to analyze due to their complex implementation or the unavailability of source code. To prevent these layers from hindering the process, we develop approximations to bound the results in such cases. Take softmax as an example, which is a monotonic function for each variable. In the case of we cannot obtain the source code, we can use the mathematical properties of the function to derive the bounds. Assume the input variables and corresponding intervals are $x_i \in [\underline{x_i}, \overline{x_i}], i = 1, \ldots, n$. We have the lower bound of x_i by

$$\frac{e^{\underline{x_i}}}{\sum e^{\overline{x_{\neq i}}} + e^{\underline{x_i}}} \geq \frac{e^{\underline{x_i}}}{\sum e^{\overline{x}}} \approx \text{softmax}(x) \times e^{(\underline{x_i} - \overline{x_i})}.$$

Similarly, we use $\text{softmax}(x) \times e^{(\overline{x_i} - \underline{x_i})}$ to approximate the upper bound. For functions that are not monotonic such as $\sin$, we use the possible maximum and minimum value to approximate the interval.

3) Efficient Error Analysis in Practice: The method described above is general enough to work with any pair of deep learning programs. However, estimating the error for each layer in the neural network can be slow in practice and error accumulation is inherent to interval methods in deep networks. RENDER mitigates this by providing layer-level granularity for error analysis. It allows users to examine any continuous layers they want so that our approach does not need to analyze the whole deep model. In real-world applications, a common scenario involves developers replacing only small parts of layers in the original programs with customized kernels to speed up, such as attention mechanisms or fused loss functions. In such cases, we only need to *start* the analysis at the first layer where the target and reference programs differ. In addition, developers may sometimes be able to localize precision issues to specific layers, rather than capturing the loss mismatches in the final output. RENDER can *end* the analysis early, before reaching the last layer of the program. RENDER can be used to analyze any continuous layers in deep learning programs, making it more practical.

IV. EVALUATION

To evaluate the effectiveness of our approach, we conduct experiments based on the following research questions.

RQ.1 Can RENDER effectively classify the root causes of errors in real-world issues about DL kernels compared to existing techniques?

RQ.2 Can RENDER scale to handle the large number of operations in deep learning programs?

RQ.3 Can RENDER work at the neural network level, i.e., the whole DL program?

In the first two questions, we focus on the effectiveness and efficiency of kernel-level round-off error estimation, while in the last question, we evaluate the performance within the context of neural network models.

A. Experimental Setup

1) Dataset: We construct our evaluation dataset by collecting real-world issues from the Triton community[3]. Triton consists of a Python-based programming language and compiler infrastructure designed for writing high-performance custom kernels [11]. It has gained widespread adoption in the deep learning community. Developers typically implement custom kernels in Triton, such as matrix multiplication, and compare the results against PyTorch [12] implementations based on CUDA. When the differences exceed expected tolerances or their expectations, they often report issues to the community, which we use to build our dataset.

We searched GitHub issues using the keywords "error", "accurate" and "precision", and filtered out cases unrelated to numerical mismatches. We collect the code, input tensors, test code, and atol (if it exists) from the issues. We initially selected 92 issues from Triton communities. After filtering out those unrelated to numerical problems (e.g., build failures, performance issues), 25 cases remained. Issue #3013 is a duplicate of #3017. Issues #3478, #2680a, #1493, and #4113 cannot be reproduced due to specific hardware or environmental requirements. As a result, 20 test cases related to precision issues comprise our dataset. For closed issues, we manually labeled the root cause of each error as either Type-I (round-off error) or Type-II (implementation or compiler bugs) based on the discussions and final resolutions. For the open issues, we first analyze the corresponding issues and the comments carefully, and resolve them in local environments based on RENDER. Then, we posed the comments and solutions on the issues to communicate with the developers and seek their opinions. Finally, we labeled them based on the discussions and communications. As a result, 16 cases were classified as Type-I, and 4 as Type-II.

2) Implementation: Our implementation is built upon Triton [11], which provides a highly efficient platform for writing custom deep learning primitives. We modified the Triton interpreter by integrating interval arithmetic for each tensor operation. Specifically, we track the interval of each tensor as the kernel executes, ensuring that both the lower and upper bounds of each tensor are updated at each operation. All operations are implemented in a vectorized form utilizing numpy, ensuring compatibility with batch computations and efficiency. We use double precision for the lower and upper bounds of the interval and the corresponding machine epsilon for the error estimation. This enables us to model the propagation of rounding errors throughout the computation accurately. In addition, we use numba[4] to accelerate the computation of the interval arithmetic, which significantly improves the performance of our approach.

[3]https://github.com/triton-lang/triton/issues

[4]https://numba.pydata.org/

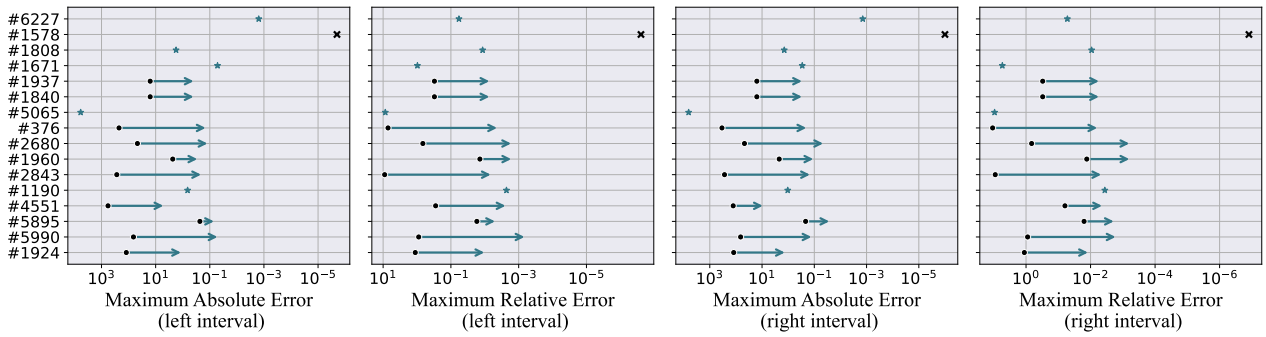


Fig. 4: Comparison of the output tensor bound and the max difference between RENDER and SATIRE. For the cases where SATIRE fails, we represent the interval bounds using "*" symbol. For the cases where RENDER and SATIRE yield identical results, we use an "x" symbol to represent them. A smaller error (right side) is better.

3) Baseline: We consider the following baselines.

- SATIRE [15]. As RENDER is the first work to classify the errors in deep learning programs by estimating the round-off error, we compare it with an approach that aims to estimate the round-off error for floating-point computations. SATIRE [15] is a scalable and rigorous error estimation tool based on symbolic Taylor expansion. It is especially designed for a large number of operations by several heuristics, such as abstraction and path reduction, so it is an appropriate baseline for RENDER. Since SATIRE does not support mixed precision, we use the lowest possible precision in each test case to ensure the soundness of its estimated bounds. If the time required for analysis is less than an hour, we use the default configuration. However, if the analysis time exceeds one hour, we switch to using abstraction techniques to speed up the process. As SATIRE does not rely on Triton, we construct an equivalent program in its format to facilitate comparison with our approach.
- High-Precision Re-execution. We also consider a straightforward baseline where the program is re-executed with higher precision. If the errors persists under higher precision, it is attributed to implementation bugs; otherwise, it is classified as round-off errors.

We do not consider approaches that aim to detect numerical bugs in neural networks, such as DEBAR [27], as a baseline and the detail comparision is left in Section VI. All experiments are conducted on a machine running Ubuntu 20.04 with an A800 GPU and 64GB of memory.

B. Results

1) RQ1: Effectiveness: To answer RQ1, we apply RENDER and baselines to the dataset. For each issue, we rerun the target kernel and estimate the interval of the output tensor. If the reference output falls within the interval, we classify the error as Type-I; otherwise, we classify it as Type-II. The overall results are summarized in Table III. The results demonstrate that RENDER successfully classifies all issues in the dataset.

TABLE III: Overall results of our approach and baselines.

Issue ID	GT	RENDER	SATIRE	High Precision
#1924	Type-I	✓	✓	✓
#5990	Type-I	✓	✓	✓
#3017	Type-II	✓	✓	✓
#5895	Type-I	✓	✓	✓
#4551	Type-I	✓	✗	✓
#1190	Type-I	✓	Fail	✓
#2843	Type-I	✓	✗	✓
#1960	Type-I	✓	✓	✓
#2680	Type-I	✓	✓	✗
#376	Type-I	✓	✓	✓
#5065	Type-I	✓	Not Supported	✓
#1840	Type-I	✓	✓	✓
#1666	Type-II	✓	✓	✓
#1937	Type-I	✓	✓	✗
#1671	Type-I	✓	Timeout	✓
#1821	Type-II	✓	Timeout	Not Supported
#4701	Type-II	✓	✓	✓
#1808	Type-I	✓	Timeout	✗
#1578	Type-I	✓	✓	Not Supported
#6227	Type-I	✓	Timeout	✓
Total		**20**	12	15

Compared to RENDER, SATIRE only successfully classifies 12 out of 20 issues. The remaining 8 issues either timeout, do not support FP8 floating-point format, or fail to provide a sound interval. High-Precision Re-execution classifies 15 issues. It indicates that RENDER is more effective in classifying the root cause of errors in deep learning kernels. We provide the detailed comparison between RENDER and the baselines in the following sections.

Comparison with SATIRE. To further compare the tightness of the bounds with SATIRE, we visualize the absolute and relative errors of the output tensor with intervals estimated by the two approaches for 16 Type-I error cases in Fig. 4. For the cases where both SATIRE and RENDER successfully run, we observe that our approach always provides a tighter interval

than SATIRE. In some instances, SATIRE yields a relative error of 10, while RENDER reduces it to 0.1, achieving a maximum improvement of up to 100x. The main reason for the overestimation of SATIRE is mixed precision, and we leave the detailed analysis to the discussion.

Comparison with High-Precision Re-execution. Compared with our approach, High-Precision Re-execution can correctly classify 15 issues. In particular, it works well when numerical errors are dominated by round-off errors and can be mitigated by simply re-running the program with higher precision. While it is effective for a subset of cases, it has several limitations. First, it cannot handle scenarios where the program is already executed in high precision (e.g., #1821, #1578). Second, round-off error between the reference implementation and the target program may still exceed acceptable thresholds even when using higher precision (e.g., #1937, #1808), which can lead to misclassification of Type-I errors as Type-II. Finally, low-level code generation and optimization strategies often vary across precisions, which can alter high-precision execution results and potentially mask underlying issues. (e.g., #2680).

In addition, this approach does not necessarily help developers gain confidence in adopting low-precision implementations, regardless of the high-precision behavior. For example, in issue #4551, the developer compared four versions of a kernel using either f32 or TF32 in Triton and PyTorch. While the errors with f32 were acceptable, those with TF32 were considered unacceptable. The developers remained uncertain about TF32's behavior and suspected potential bugs. The experimental results and potential concerns of high-precision execution further indicate that RENDER is more effective in classifying the root cause of errors in deep learning kernels.

> **Answer to RQ1:** RENDER can successfully classify all the issues in the dataset, while SATIRE and High-Precision Re-execution classify 12 and 15 out of 20 issues, respectively. In addition, RENDER can always provide tighter interval estimations compared to SATIRE.

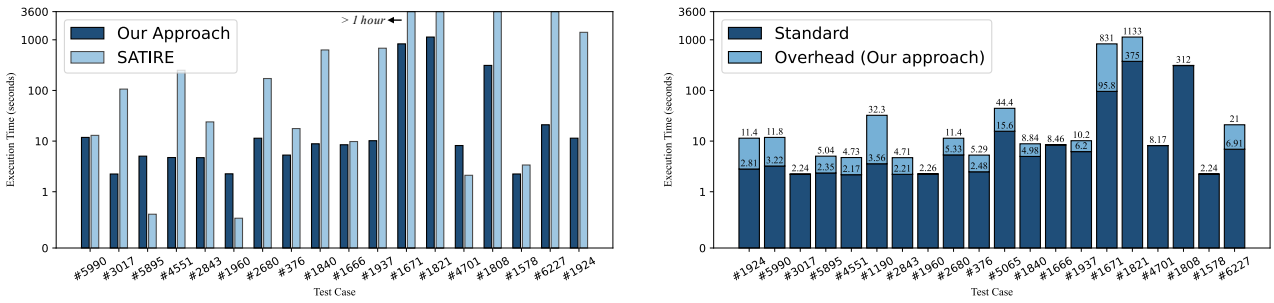
2) RQ2: Efficiency: To answer RQ2, we compare the time costs of RENDER with SATIRE and evaluate the runtime overhead. We do not list the results of High-Precision Re-execution, as the running time of higher precision is similar to that of the original program and our focus is the overhead of interval error estimation. As shown in Fig. 5a, our RENDER exhibits significantly lower runtime compared to SATIRE. Considering the average across all cases, our approach achieves a 19× speedup over SATIRE. While SATIRE is designed for large-scale numerical programs, it encounters timeouts in 25% of the test cases. In contrast, our approach successfully handles all test cases and runs faster than SATIRE in most cases. In the three cases where SATIRE runs faster than our approach, the execution times are all less than ten seconds. This is because the initial startup time of RENDER is longer due to the involvement of the Triton runtime. We argue that these cases are of little practical significance. These experimental results demonstrate that RENDER is not only more efficient but also more practical than SATIRE for real-world computations.

In Fig. 5b, we measure the runtime overhead by comparing the execution time of the target program with and without interval error estimation. RENDER incurs an average overhead of 2.7x, with a maximum of 9x. While the overhead of our approach is noticeable, it remains manageable given the efficiency gains achieved through its practical application in real-world scenarios. Furthermore, the overhead is relatively consistent across test cases, making it predictable and scalable.

> **Answer to RQ2:** RENDER runs faster 19x than SATIRE and incurs a runtime overhead of 2.7x on average, which indicates its efficiency in practice.

3) RQ3: Neural Network level Classification: To answer RQ3, we evaluate RENDER on the DL programs collected from real-world issues in the PyTorch community. Following the same data collection process, we select 7 cases. We first localize the layers based on the issues and conduct the same kernel-level analysis, as discussed in Section III-C. Since several cases involve C++, we also implement the prototype of our approach for them. As a result, 6 of 7 cases are classified successfully, which shows that RENDER can still provide an error bound to help developers understand the root cause of the error for the neural network. For the gradient accumulation cases mentioned in the introduction, RENDER can bound the error by a much tighter interval. The key point is that the only difference lies in the final `cross_entropy` function. Instead of estimating loss from the first layer, where the results are generally uncontrollable, RENDER starts on the last layer, which indicates the effectiveness of our technique. RENDER fails on the `tanh` cases, where the developer uses a hand-written implementation for `tanh` function, i.e. $\frac{2}{1+e^{-2x}} - 1$. It causes a much smaller error, which is amplified by the following `atan2` functions. The interval estimated by our approach is wider than the error bounds, and we decide it as a Type-I error wrongly. To overcome this limitation, we plan to incorporate error propagation analysis that accounts for the sensitivity of downstream functions.

False Positives and Negatives. In principle, our approach always provides an error bound that is no smaller than the actual round-off error, ensuring that if the error is indeed Type-I (round-off error), it can always be classified correctly. A detailed discussion of potential threats to soundness is provided in Section V-C2. Our experiments further provide empirical evidence of this property: we did not observe any false positives, i.e., cases where our approach classifies an error as Type-II when it is actually Type-I. It also indicates that our approach is effective in identifying and classifying round-off errors in practice. In contrast, false negatives, i.e. cases where our approach classifies an error as Type-I when it is actually Type-II, can occasionally occur. Although implementation bugs typically introduce errors far larger than round-off errors, there are situations where the resulting errors are small enough to fall within the estimated intervals, thereby



(a) Comparing time costs of RENDER and SATIRE.

(b) Runtime overhead of RENDER.

Fig. 5: Time costs comparison of RENDER.

failing to capture the bug, e.g. the `tanh` case mentioned above. Nevertheless, such cases were very limited in our evaluation, indicating that the trade-off between precision and recall is acceptable. We acknowledge that certain implementation bugs may not be triggered by the given inputs, potentially leading to additional false negatives. These latent bugs are beyond the scope of our approach, which is intended to support developers once observable mismatches arise. In conclusion, our method provides a practical means for analyzing numerical errors, while acknowledging the possibility of false negatives.

Answer to RQ3: RENDER is able to classify 6 of 7 cases in neural network level cases, which shows its effectiveness of our approach.

V. DISCUSSION

A. Why interval arithmetic is more effective than symbolic Taylor expansion?

It may be surprising that interval arithmetic (RENDER) outperforms symbolic Taylor expansion (SATIRE) in our experiments, both in terms of bound precision and efficiency. Symbolic Taylor expansion used by SATIRE is more accurate than interval arithmetic by our approach in theory, however, it does not give a tighter bound in practice. We found two reasons to explain it. (1) The advantage of symbolic Taylor expansion lies in its ability to utilize the relationships between variables for more accurate error estimation. However, since the variable relationships in our dataset are not complex and the errors can be considered independent, the strength of SATIRE is not fully exploited. (2) SATIRE does not support mixed precision, which may lead to overestimation of the error. For example, the machine epsilon of float16 is approximately 9.77×10^{-4}, while for float32, it is about 1.19×10^{-7}, a difference of roughly 100x. In matrix multiplication, the summation error is given by $(m-1) \times \varepsilon_{+}$, where m is the number of elements being summed. Suppose we use the larger machine epsilon of float16, the error will be overestimated by 100x, much larger than the difference between the errors predicted by Taylor expansion and interval arithmetic. In addition, we apply the same precision constraint in RENDER as in SATIRE (i.e., using the lowest precision). Experimental results show that our approach correctly classified 19 out of 20 cases. For issue #1821, which is in fact a compiler error, RENDER misclassified it as a Type-I (round-off error). This misclassification was caused by a wider interval resulting from not accounting for mixed-precision effects, leading to a false negative. These underscore the importance of properly handling mixed precision in practice.

B. Case Study and Developer Feedback

To better understand the practical impact of our approach, we conducted a case study and examined 7 open issues. Excluding 2 older cases from 2023, we provided comments and potential solutions for the remaining 5. Among these, we received feedback on 3: two developers responded positively and agreed with our analysis and solutions, while another developer remained uncertain and considered alternative explanations. These results suggest that RENDER can effectively assist developers in understanding the root causes of many errors in practice. We select three representative issues from the dataset to demonstrate the classification results.

1) Case 1: Round-off Error: The first example is about matrix multiplication in issue #1808, where the error is due to the round-off error but is considered a bug by the developer. The code snippet is shown below.

```
1   A += ram * stride_am + rk * stride_ak
2   B += rk * stride_bk + rbn * stride_bn
3   acc = tl.zeros((BLOCK_M, BLOCK_N), dtype=ACC_TYPE)
4   for k in range(K, 0, -BLOCK_K * SPLIT_K):
5       a = tl.load(A, mask=rk[None, :] < k, other=0.0)
6       b = tl.load(B, mask=rk[:, None] < k, other=0.0)
7       acc += tl.dot(a, b)
8       A += BLOCK_K * SPLIT_K * stride_ak
9       B += BLOCK_K * SPLIT_K * stride_bk
10  acc = acc.to(C.dtype.element_ty)
11  rm = pid_m * BLOCK_M + tl.arange(0, BLOCK_M)
12  rn = pid_n * BLOCK_N + tl.arange(0, BLOCK_N)
13  C += rm * stride_cm + rn * stride_cn
14  if SPLIT_K == 1:
15      tl.store(C, acc)
16  else:
17      tl.atomic_add(C, acc)
```

Lines 1-13 represent typical matrix multiplication code that calculates the offset, loads the data, and performs the computation. When `SPLIT_K` is set to 1, the output tensor is

directly updated by `tl.store`, otherwise, the output tensor is updated by `tl.atomic_add` (Line 14-17) by different threads atomically. However, the summation order is not fixed and is determined in runtime, which can lead to minor differences in the results. In practice, the maximum difference can be 0.1875, which the developer initially considered a bug. Upon reviewing the code, we determined that the issue arises from round-off errors in the addition operations and communicated this to the developer.

It is worth noting that after we comment on the issue with our analysis results and the above explanation, the developer acknowledges the round-off error and confirms the possibility. It indicates that RENDER can help developers understand the root cause of the error in practice.

2) Case 2: Compiler Bug: The second example reveals a compiler bug that causes incorrect results in attention computation in issue #1821. The issue stems from an incorrect calculation of the number of unique threads within a warp—the compiler failed to properly account for the number of elements processed per thread. This leads to incorrect behavior in certain cases, resulting in a maximum difference of 0.15 in the output. In this case, RENDER concludes that the error is due to a Type-II error as $0.23 \notin [0.32, 0.37]$, i.e., the maximum difference is outside the interval. This bug occurs only in large-scale computations, and SATIRE times out on this issue, so it cannot detect it. Our approach can handle this issue in 12 minutes.

3) Case 3: Implementation Bug: The last example in issue #1666 illustrates an implementation bug in code that aims to compute the matrix multiplication between the transpose of tensor A and B. However, the developer mistakenly uses `stride_ak` and `stride_am` in the offset calculation, as shown below. The code snippet is shown below.

```
- a_ptrs = a_ptr + (offs_k[:, None] * stride_ak
-   + offs_am[None, :] * stride_am)
+ a_ptrs = a_ptr + (offs_k[:, None] * stride_am
+   + offs_am[None, :] * stride_ak)
```

In this case, the element with the maximum difference is approximately -0.6, which does not fall within the interval [10.001, 10.079] estimated by our approach, indicating a clear bug. Interestingly, the developer did not identify this issue at first, and it was only after a thorough review by the maintainer of Triton that the bug was discovered. This highlights the difficulty of debugging such issues, and our approach can play a crucial role in the first step of classifying the root cause.

C. Threats to Validity

1) External Validity: For external validity, it arises from the dataset collection and labeling process, as we manually curated and labeled the issues from the Triton and PyTorch communities. This introduces the possibility of bias in the dataset, particularly in terms of how we categorize and interpret the issues related to precision and error. To mitigate this bias, we reproduce each reported issue and engage in discussions with developers for open issues to better understand the root causes. Furthermore, the issues we selected for our dataset of the kernel-level come from a single deep learning framework (Triton), which may limit the generalizability of RENDER to other frameworks or domains. As Triton is widely used in deep learning communities, we believe that the insights from this dataset can be valuable for identifying and classifying precision errors in other deep learning frameworks as well.

2) Internal Validity: The major threat to internal validity is the soundness on interval arithmetic for error estimation. We use double-precision for the intervals rather than arbitrary-precision libraries such as mpmath, which may affect the soundness of our approach. However, this would slow down runtime, as to the best of our knowledge, such libraries do not benefit from GPU acceleration significantly. Double-precision is already sufficiently accurate when compared to widely used formats such as TF32 and BF16 in deep learning. Therefore, we chose double precision in our implementation as a balance between soundness and efficiency. In addition, our approach might underestimate the interval, as ULP bounds listed in TABLE II are not guaranteed in CUDA [23]. We believe that such cases are rare, as the inputs used by developers are typically randomly generated, rather than carefully designed to trigger numerical instability.

D. Limitation and Future Work

Binary Classification. The main limitation of RENDER is that it only classifies errors into two types: Type-I and Type-II. While this classification serves as a useful starting point for identifying the nature of floating-point errors, it is not sufficient for a detailed root cause analysis. Developers may still need to conduct additional investigations into the code to pinpoint the exact cause of the error, especially in more complex cases where multiple contributing factors might be involved. In future work, we plan to explore more advanced classification techniques that can move beyond the two-class system. In addition to considering more types of errors, techniques such as machine learning-based classification or hierarchical error classification may allow for finer-grained analysis and a deeper understanding of the underlying causes of floating-point errors. This would help developers better diagnose errors that involve more complex interactions between operations in large-scale deep learning models.

Manual Effort. A limitation of RENDER is the manual effort required to define interval semantics for each floating-point operator. Although the set of operators in deep learning frameworks is finite, their numerical behaviors are often non-trivial to model. To mitigate this challenge, we outline several directions. First, most operators can be grouped into a few categories, such as linear algebra kernels, element-wise arithmetic, and reduction operations. For each category, interval propagation rules can be systematically derived and reused, rather than implemented individually. Second, existing operator specifications (e.g., ONNX or MLIR dialects) can be leveraged to automatically generate interval semantics, thereby reducing the need for manual coding. Finally, in deep learning, many complex mathematical functions are rarely used, and implementing interval semantics for the most common operators is sufficient to handle the majority of real-world

workloads. These strategies suggest that developing practical interval semantics for deep learning operators is both feasible and scalable, despite the complexity of individual cases.

Limited Scope. Another limitation is that the scope of the evaluation focused mainly on Triton and PyTorch framework. We believe our approach can be easily generalized to other frameworks as the core tensor interval abstraction such as arithmetic operation is shared and framework independent. With reasonable engineering effort, it can be adapted to accommodate the different APIs used by various frameworks.

VI. Related Work

A. Estimation and Testing of Floating-Point Error

As the floating-point error is inevitable, the estimation of the maximum floating-point error has been studied extensively in traditional numerical computing [21], [28], [25]. Interval arithmetic [21] approximates rounding errors by maintaining the lower and upper bounds, ensuring the true result lies within the computed bounds. However, it cannot model the relations of variables, which may cause false positives. Since interval arithmetic cannot capture the correlations between inputs, affine arithmetic [28] has been proposed to refine these bounds by representing the error as an affine form. Rosa [29] uses SMT solvers to generate a finite-precision implementation guaranteed to meet the desired precision for real numbers. For more accurate error estimation in higher-order terms, Solovyev et al. [25] propose a tool called FPTaylor, based on symbolic Taylor expansions, which has been used to verify the mixed precision synthesis tool FPTuner [30]. SATIRE [15] tries to scale the Taylor expansion to large-scale programs based on several heuristics. Although Taylor expansion-based approaches can offer more accurate error estimates, their computational cost is often prohibitive. In contrast, our method focuses on error analysis in DL programs and provides fast feedback to developers. While our approach may not be as theoretically accurate as symbolic Taylor expansion, such as SATIRE, it is more efficient for deep learning programs.

Other works adopt a testing-based perspective, aiming to search for inputs that can trigger larger errors, such as Eiffel [31]. Zou et al. use atomic condition to effectively guide the search for large floating-point errors [32], which depends on the condition of single floating-point operations. However, atomic condition is insufficient and does not provide a comprehensive solution for all scenarios. Based on the above work, FPCC generalizes the atomic condition to chain conditions [33], which guides the search more sophisticatedly. After the error is estimated, the next step is to repair the identified issues and improve the numerical stability of the program. Herbie [34] and Salsa [35] improve the numerical stability of floating-point programs by automatically rewriting the code. Zou et al. [36] propose an oracle-free method to repair floating-point programs dynamically.

B. Analysis and Testing of Deep Learning Programs

Instead of focusing on round-off error in floating-point numbers, another line of our related work focuses on the analysis and testing of deep learning programs and operators. Predoo [37] is an early work in precision testing of DL programs, which treats the testing process as a search problem and aims to find inputs that lead to large precision errors. Duo [38] uses differential testing to compare the outputs of different model implementations on the same input and identifies inconsistencies that may indicate bugs. Chen et al. [39] apply metamorphic testing, checking whether certain relationships between inputs and outputs always hold.

Zhang et al. [27] propose DEBAR, a tool to detect numerical errors in deep learning programs based on interval abstraction in the abstract interpretation framework [40]. While our approach and DEBAR both use intervals, the key differences between our approach and DEBAR lie in their goals: RENDER aims to explain mismatches to a reference implementation, whereas DEBAR aims to explore numerical bugs(e.g., NAN) for given programs. Our approach uses interval arithmetic to dynamically estimate the potential round-off errors and classify the root cause. In contrast, DEBAR uses interval abstraction to statically estimate the lower and upper bounds, then detects unsafe operations based on the interval. RENDER can be used to identify potential bugs by classifying them as Type-II, i.e., implementation bugs. These bugs often involve general semantic issues rather than purely numerical errors, typically arising from compilation faults or from operators that fail to implement the intended functionality of the reference implementation. Since DEBAR focuses exclusively on numerical errors, it is not suitable for detecting such semantic mismatches; therefore, we did not include it as a baseline in our evaluation.

VII. Conclusion

Deep learning programs are becoming increasingly popular, with many different optimizations available for the same formula for performance purposes. Developers often face challenges in debugging when the outputs of two implementations differ. In this work, we present a lightweight, dynamic interval analysis framework for estimating round-off errors in DL programs and classifying the errors based on the intervals. Our method offers practical and effective error estimation by executing the target program and tracking the interval bounds of intermediate tensors. Compared to the baseline approach, our method accurately classifies numerical errors and helps identify stability issues in real-world DL programs. Furthermore, the experimental results show that the runtime overhead of our approach is acceptable and does not undermine its practicality. Overall, our approach bridges the gap between theory and practice in round-off error estimation, improving the reliability of deep learning systems.

Our implementation and dataset are available at https://github.com/Qi-Zhan/Render.

Acknowledgments

This research is supported by the National Key R&D Program of China (No. 2024YFB4506400) and CCF-Huawei Populus Grove Fund.

References

[1] K. Bayoudh, R. Knani, F. Hamdaoui, and A. Mtibaa, "A survey on deep multimodal learning for computer vision: advances, trends, applications, and datasets," *The Visual Computer*, vol. 38, pp. 2939 – 2970, 2021. [Online]. Available: https://api.semanticscholar.org/CorpusID:235410640

[2] D. W. Otter, J. R. Medina, and J. K. Kalita, "A survey of the usages of deep learning for natural language processing," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 32, no. 2, pp. 604–624, 2021.

[3] Y. Yang, X. Xia, D. Lo, and J. Grundy, "A survey on deep learning for software engineering," *ACM Comput. Surv.*, vol. 54, no. 10s, Sep. 2022. [Online]. Available: https://doi.org/10.1145/3505243

[4] T. Dao, D. Y. Fu, S. Ermon, A. Rudra, and C. Ré, "FlashAttention: Fast and memory-efficient exact attention with IO-awareness," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

[5] M. Wu, X. Cheng, S. Liu, C. Shi, J. Ji, K. Ao, P. Velliengiri, X. Miao, O. Padon, and Z. Jia, "Mirage: A multi-level superoptimizer for tensor programs," in *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*. Boston, MA: USENIX Association, Jul. 2025.

[6] A. Farkas, G. Kertész, and R. Lovas, "Parallel and distributed training of deep neural networks: A brief overview," in *2020 IEEE 24th International Conference on Intelligent Engineering Systems (INES)*, 2020, pp. 165–170.

[7] S. Li, Y. Zhao, R. Varma, O. Salpekar, P. Noordhuis, T. Li, A. Paszke, J. Smith, B. Vaughan, P. Damania, and S. Chintala, "Pytorch distributed: experiences on accelerating data parallel training," *Proc. VLDB Endow.*, vol. 13, no. 12, p. 3005–3018, Aug. 2020. [Online]. Available: https://doi.org/10.14778/3415478.3415530

[8] L. Zheng, Z. Li, H. Zhang, Y. Zhuang, Z. Chen, Y. Huang, Y. Wang, Y. Xu, D. Zhuo, E. P. Xing, J. E. Gonzalez, and I. Stoica, "Alpa: Automating inter- and Intra-Operator parallelism for distributed deep learning," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. Carlsbad, CA: USENIX Association, Jul. 2022, pp. 559–578. [Online]. Available: https://www.usenix.org/conference/osdi22/presentation/zheng-lianmin

[9] P. Micikevicius, S. Narang, J. Alben, G. F. Diamos, E. Elsen, D. García, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh, and H. Wu, "Mixed precision training," in *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018. [Online]. Available: https://openreview.net/forum?id=r1gs9JgRZ

[10] P. Micikevicius, D. Stosic, N. Burgess, M. Cornea, P. Dubey, R. Grisenthwaite, S. Ha, A. Heinecke, P. Judd, J. Kamalu, N. Mellempudi, S. Oberman, M. Shoeybi, M. Siu, and H. Wu, "Fp8 formats for deep learning," 2022. [Online]. Available: https://arxiv.org/abs/2209.05433

[11] P. Tillet, H. T. Kung, and D. Cox, "Triton: an intermediate language and compiler for tiled neural network computations," in *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, ser. MAPL 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 10–19. [Online]. Available: https://doi.org/10.1145/3315508.3329973

[12] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, *PyTorch: an imperative style, high-performance deep learning library*. Red Hook, NY, USA: Curran Associates Inc., 2019.

[13] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, J. Davison, S. Shleifer, P. von Platen, C. Ma, Y. Jernite, J. Plu, C. Xu, T. L. Scao, S. Gugger, M. Drame, Q. Lhoest, and A. M. Rush, "Transformers: State-of-the-art natural language processing," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Online: Association for Computational Linguistics, Oct. 2020, pp. 38–45. [Online]. Available: https://www.aclweb.org/anthology/2020.emnlp-demos.6

[14] A. Zucker. Fix gradient accumulation issue. Hugging Face. GitHub Pull Request #34191. [Online]. Available: https://github.com/huggingface/transformers/pull/34191

[15] A. Das, I. Briggs, G. Gopalakrishnan, S. Krishnamoorthy, and P. Panchekha, "Scalable yet rigorous floating-point error analysis," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '20. IEEE Press, 2020.

[16] "Ieee standard for floating-point arithmetic," *IEEE Std 754-2008*, pp. 1–70, 2008.

[17] N. Burgess, J. Milanovic, N. Stephens, K. Monachopoulos, and D. Mansell, "Bfloat16 processing for neural networks," in *2019 IEEE 26th Symposium on Computer Arithmetic (ARITH)*, 2019, pp. 88–91.

[18] P. Valero-Lara, I. Jorquera, F. Lui, and J. Vetter, "Mixed-precision s/dgemm using the tf32 and tf64 frameworks on low-precision ai tensor cores," in *Proceedings of the SC '23 Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis*, ser. SC-W '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 179–186. [Online]. Available: https://doi.org/10.1145/3624062.3624084

[19] N. J. Higham, *Accuracy and Stability of Numerical Algorithms*. USA: Society for Industrial and Applied Mathematics, 1996.

[20] D. Goldberg, "What every computer scientist should know about floating-point arithmetic," *ACM Comput. Surv.*, vol. 23, no. 1, p. 5–48, Mar. 1991. [Online]. Available: https://doi.org/10.1145/103162.103163

[21] R. E. Moore, *Interval Analysis*. Englewood Cliffs, NJ: Prentice-Hall, 1966.

[22] "Ieee standard for interval arithmetic," *IEEE Std 1788-2015*, pp. 1–97, 2015.

[23] NVIDIA, "Cuda c programming guide," 2023. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[24] R. Nobre, L. Reis, J. a. Bispo, T. Carvalho, J. a. M. Cardoso, S. Cherubin, and G. Agosta, "Aspect-driven mixed-precision tuning targeting gpus," in *Proceedings of the 9th Workshop and 7th Workshop on Parallel Programming and RunTime Management Techniques for Manycore Architectures and Design Tools and Architectures for Multicore Embedded Computing Platforms*, ser. PARMA-DITAM '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 26–31. [Online]. Available: https://doi.org/10.1145/3183767.3183776

[25] A. Solovyev, M. S. Baranowski, I. Briggs, C. Jacobsen, Z. Rakamarić, and G. Gopalakrishnan, "Rigorous estimation of floating-point round-off errors with symbolic taylor expansions," *ACM Trans. Program. Lang. Syst.*, vol. 41, no. 1, Dec. 2018. [Online]. Available: https://doi.org/10.1145/3230733

[26] M. Fasi, N. J. Higham, M. Mikaitis, and S. Pranesh, "Numerical behavior of nvidia tensor cores," *PeerJ Computer Science*, vol. 7, p. e330, 2021. [Online]. Available: https://doi.org/10.7717/peerj-cs.330

[27] Y. Zhang, L. Ren, L. Chen, Y. Xiong, S.-C. Cheung, and T. Xie, "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*, ser. ESEC/FSE 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 826–837. [Online]. Available: https://doi.org/10.1145/3368089.3409720

[28] L. H. de Figueiredo and J. Stolfi, "Affine arithmetic: Concepts and applications," *Numerical Algorithms*, vol. 37, no. 1, pp. 147–158, Dec 2004. [Online]. Available: https://doi.org/10.1023/B:NUMA.0000049462.70970.b6

[29] E. Darulova and V. Kuncak, "Sound compilation of reals," in *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 235–248. [Online]. Available: https://doi.org/10.1145/2535838.2535874

[30] W.-F. Chiang, M. Baranowski, I. Briggs, A. Solovyev, G. Gopalakrishnan, and Z. Rakamarić, "Rigorous floating-point mixed-precision tuning," in *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 300–315. [Online]. Available: https://doi.org/10.1145/3009837.3009846

[31] Z. Zhang, B. Zhou, J. Hao, H. Yang, M. Cui, Y. Zhou, G. Song, F. Li, J. Xu, and J. Zhao, "Eiffel: Inferring input ranges of significant floating-point errors via polynomial extrapolation," in *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '23. IEEE Press, 2024, p. 1441–1453. [Online]. Available: https://doi.org/10.1109/ASE56229.2023.00139

[32] D. Zou, M. Zeng, Y. Xiong, Z. Fu, L. Zhang, and Z. Su, "Detecting floating-point errors via atomic conditions," *Proc. ACM Program. Lang.*, vol. 4, no. POPL, Dec. 2019. [Online]. Available: https://doi.org/10.1145/3371128

[33] X. Yi, H. Yu, L. Chen, X. Mao, and J. Wang, "Fpcc: Detecting floating-point errors via chain conditions," *Proc. ACM Program. Lang.*, vol. 8, no. OOPSLA2, Oct. 2024. [Online]. Available: https://doi.org/10.1145/3689764

[34] P. Panchekha, A. Sanchez-Stern, J. R. Wilcox, and Z. Tatlock, "Automatically improving accuracy for floating point expressions," *SIGPLAN Not.*, vol. 50, no. 6, p. 1–11, Jun. 2015. [Online]. Available: https://doi.org/10.1145/2813885.2737959

[35] N. Damouche and M. Martel, "Salsa: An automatic tool to improve the numerical accuracy of programs," in *AFM@ NFM*, 2017, pp. 63–76.

[36] D. Zou, Y. Gu, Y. Shi, M. Wang, Y. Xiong, and Z. Su, "Oracle-free repair synthesis for floating-point programs," *Proc. ACM Program. Lang.*, vol. 6, no. OOPSLA2, Oct. 2022. [Online]. Available: https://doi.org/10.1145/3563322

[37] X. Zhang, N. Sun, C. Fang, J. Liu, J. Liu, D. Chai, J. Wang, and Z. Chen, "Predoo: precision testing of deep learning operators," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 400–412. [Online]. Available: https://doi.org/10.1145/3460319.3464843

[38] X. Zhang, J. Liu, N. Sun, C. Fang, J. Liu, J. Wang, D. Chai, and Z. Chen, "Duo: Differential fuzzing for deep learning operators," *IEEE Transactions on Reliability*, vol. 70, no. 4, pp. 1671–1685, 2021.

[39] J. Chen, C. Jia, Y. Yan, J. Ge, H. Zheng, and Y. Cheng, "A miss is as good as a mile: Metamorphic testing for deep learning operators," *Proc. ACM Softw. Eng.*, vol. 1, no. FSE, Jul. 2024. [Online]. Available: https://doi.org/10.1145/3660796

[40] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL '77. New York, NY, USA: Association for Computing Machinery, 1977, p. 238–252. [Online]. Available: https://doi.org/10.1145/512950.512973