Fuzzy PyTorch: Rapid Numerical Variability Evaluation for Deep Learning Models

Anonymous authors
Paper under double-blind review

Abstract

We introduce Fuzzy PyTorch, a framework for rapid evaluation of numerical variability in deep learning (DL) models. As DL is increasingly applied to diverse tasks, understanding variability from floating-point arithmetic is essential to ensure robust and reliable performance. Tools assessing such variability must be scalable, efficient, and integrate seamlessly with existing frameworks while minimizing code modifications. Fuzzy PyTorch enables this by integrating Stochastic Arithmetic into PyTorch through Probabilistic Rounding with Instruction Set Management, a novel library interfacing with Verificarlo, a numerical analysis compiler. The library offers two modes: stochastic rounding, preserving exact floating-point operations, and up-down rounding, a faster alternative. Comparative evaluations show Fuzzy PyTorch maintains model performance while up-down rounding achieves runtime reductions of 5× to 60× versus Verrou, a state-of-the-art tool. We further demonstrate scalability by running models from 1 to 341 million parameters, confirming applicability across small and large DL architectures. Overall, Fuzzy PyTorch provides an efficient, scalable, and practical solution for assessing numerical variability in deep learning, enabling researchers and practitioners to quantify and manage floating-point uncertainty without compromising performance or computational efficiency.

1 Introduction

Many scientific domains, including medical imaging, have increasingly adopted deep learning (DL) models for computational analysis. However, these models often rely on numerical computations that are sensitive to variations in floating-point precision and rounding modes. Understanding numerical variability in DL models is therefore essential for ensuring reliable and reproducible outcomes. This is particularly critical in high-stakes applications such as medical imaging (Anonymized, b;d;f), remote sensing (Anonymized, e), and scientific simulations (Anonymized, a).

The importance of numerical variability is beginning to gain industry-wide recognition. For example, variability analysis is now supported in Amazon Web Services' (AWS) Neuron SDK (AWS Neuron Team, n.d.), AWS Trainium chips, AMD MI300 GPUs, Tesla D1s and Blackwell architecture NVIDIA GPUs (El Arar, 2025). This reflects a growing demand for tools and frameworks that enable low-level control over numerical behavior in AI systems.

Numerical variability is inherent to finite-precision systems like IEEE-754 floating-point arithmetic, which underpins most computational processes. Floating-point arithmetic introduces small rounding errors that can propagate and accumulate, ultimately affecting the reliability of outputs. This issue is especially critical in fields where reproducibility and robustness are paramount, such as neuroimaging. Assessing numerical variability is therefore essential to ensure confidence in computational results.

Although exact methods such as interval arithmetic (Hickey et al., 2001), symbolic execution (Solovyev et al., 2018), and formal verification (Boldo & Melquiond, 2011) exist to evaluate numerical accuracy, these approaches often require extensive modifications to the codebase and do not scale efficiently to complex deep learning workloads. Instead, we adopt Stochastic Arithmetic (SA), a technique that introduces controlled random perturbations to floating-point operations. This enables statistical estimation of the numerical

variability without modifying the underlying model architecture, making it more practical for large-scale applications.

SA encompasses various techniques, including Monte Carlo Arithmetic (MCA) (Parker, 1997), CESTAC (Brunet & Chatelin, 1986) and Stochastic Rounding (Forsythe, 1959). SA has been widely explored in numerical analysis, but remains underused in deep learning, despite recent studies demonstrating its potential to assess uncertainty in neural network training and inference (Faraone & Leong, 2019; Kloberdanz et al., 2022; Beuzeville et al., 2024; Arar et al., 2025). To leverage SA, researchers have developed tools such as Verificarlo (Denis et al., 2016), Verrou (Févotte & Lathuiliere, 2016) and CADNA (Jézéquel & Chesneaux, 2008).

Fuzzy PyTorch integrates SA into the PyTorch framework through a novel library named Probabilistic Rounding with Instruction Set Management (PRISM). PRISM implements two modes: stochastic rounding (SR) and up-down rounding (UD). SR preserves exact values by probabilistically rounding values based on their proximity to representable floating-point numbers, while UD provides a faster alternative by randomly rounding up or down with equal probability. Both modes are optimized using vectorized CPU instructions through the Highway library (Google, 2024a), minimizing computational overhead.

Fuzzy PyTorch seamlessly integrates with PyTorch by extending the Verificarlo compiler to pass vector instructions directly to PRISM, avoiding the serialization step required in the original implementation. This design ensures efficient execution while fully leveraging architecture-specific instruction sets. Fuzzy PyTorch instruments both scalar and vectorized floating-point operations and supports multi-threaded execution for efficient parallel computation.

In contrast, Verrou enforces a sequential execution model, limiting its performance in multi-threaded environments. For linear algebra operations, Fuzzy PyTorch relies on open-source BLAS and LAPACK libraries (Anderson et al., 1999), avoiding the constraints of proprietary alternatives like Intel MKL, which cannot be modified or instrumented due to their closed-source nature. A key advantage of Fuzzy PyTorch is that it enables numerical variability analysis with minimal changes to existing PyTorch codebases. Its parallel execution capabilities further enhance scalability across deep learning workflows.

In comparative evaluations against state-of-the-art tools, including Verificarlo, Verrou, CADNA, and the stochastic rounding library by Fasi and Mikaitis (Fasi & Mikaitis, 2021), Fuzzy PyTorch achieves similar numerical error characteristics while substantially reducing execution time. This performance establishes it as a practical and efficient framework for analyzing the impact of floating-point computations on deep learning outputs.

Numerical variability is inherently tied to reproducibility issues in deep learning. Variations in hardware, compiler settings, floating-point formats, or computing environments can introduce inconsistencies in floating-point operations, potentially affecting results for sensitive applications. By integrating numerical variability analysis directly into the PyTorch ecosystem, Fuzzy PyTorch provides researchers with a standalone framework to evaluate these effects, fostering more reproducible DL research.

This work proposes three main contributions:

1. **PRISM:** We introduce PRISM, which implements probabilistic rounding methods for the systematic analysis of floating-point errors.

2. **SA in PyTorch:** We seamlessly integrate stochastic arithmetic into PyTorch, enabling efficient and transparent instrumentation of deep learning operations.

3. **Comparative evaluation with Verrou:** We perform a comprehensive evaluation against Verrou, a state-of-the-art tool for numerical variability analysis, showcasing the enhanced performance and flexibility of Fuzzy PyTorch on use cases ranging from digit classification with MNIST, whole brain segmentation with the FastSurfer neuroimaging CNN and Parkinson's prediction using speech data with the WavLM model.

2 Numerical Variability Estimation

Measuring numerical variability in DL models requires two key components: 1) a method to estimate floating-point errors, and 2) metrics to quantify the resulting variability.

For the first component, we use Stochastic Arithmetic (SA), a family of techniques that introduce randomness into floating-point computations. These methods rely on non-deterministic rounding. Unlike standard IEEE-754 rounding modes, this approach rounds to either of the two closest floating-point numbers based on computed probabilities. The specific stochastic arithmetic technique is determined by how this probability is computed. In the literature, there are multiple terms that refer to stochastic arithmetic, including Monte Carlo Arithmetic (MCA), Stochastic Rounding, and CESTAC (Brunet & Chatelin, 1986). We use the umbrella term Probabilistic Rounding (PR) to refer to this class of techniques.

For the second component, we use the significant bits metric to quantify the numerical error arising from multiple executions with stochastic arithmetic.

2.1 Probabilistic Rounding

Let $\mathcal{F}$ be the set of normalized binary floating-point numbers with elements $x = (-1)^s.m.2^e$, where $s \in \{0, 1\}$ is the sign bit, $2^{p-1} \leq m < 2^p$ is the significand, $e \in \mathbb{Z}$ is the exponent and p the precision. Let the rounding functions $\lceil x \rceil : \mathbb{R} \to \mathcal{F}$ and $\lfloor x \rfloor : \mathbb{R} \to \mathcal{F}$ be such that $\lfloor x \rfloor = \max\{y \in \mathcal{F} \,|\, y \leq x\}$ and $\lceil x \rceil = \min\{y \in \mathcal{F} \,|\, y \geq x\}$, which return the previous and next representable floating point numbers to x. Probabilistic Rounding can be defined as:

$$\circ_{\text{PR}}(x, p_\circ) = \begin{cases} \lfloor x \rfloor & \text{with probability } p_\circ \\ \lceil x \rceil & \text{with probability } 1 - p_\circ \end{cases} \tag{1}$$

where $p_\circ : \mathbb{R} \to [0, 1]$

The existing stochastic arithmetic techniques can then be reinterpreted with the PR definition.

2.1.1 IEEE-754 Rounding

The IEEE-754 round-to-nearest mode, also known as round-to-nearest, ties-to-even, is the default rounding mode used in most floating-point hardware and software implementations. When a real number cannot be exactly represented in floating point, it is rounded to the closest representable number. If the number lies exactly halfway between two floating-point values, the tie is broken by rounding to the one with an even least significant bit (i.e., the one whose mantissa ends in 0).

Formally, the rounding function $\circ_{\text{RN}}(x)$ maps $x \in \mathbb{R}$ to the nearest floating-point number $f \in \mathcal{F}$ such that:

$$\circ_{\text{RN}}(x) = \arg\min_{f \in \mathbb{F}} |x - f|$$

with ties resolved to the floating-point number f such that the significand of f is even. This method minimizes rounding bias over repeated operations and is the most widely adopted deterministic rounding strategy defined by the IEEE-754 standard (IEEE Computer Society, 2008).

2.1.2 CESTAC Rounding

The Control of Accuracy and Debugging for Numerical Applications (CESTAC) technique simulates the roundoff error in floating-point arithmetic by rounding upward or downward the result of each floating-point operation with equal probabilities.

$$\circ_{\text{CESTAC}}(x) = \begin{cases} \lfloor x \rfloor & \text{with probability } \frac{1}{2} \\ \lceil x \rceil & \text{with probability } \frac{1}{2} \end{cases} \tag{2}$$

CESTAC is implemented in the CADNA library (Jézéquel & Chesneaux, 2008).

2.1.3 Stochastic Rounding

The Stochastic Rounding (Forsythe, 1959) (SR) technique rounds the result of each floating-point with a probability that depends on the distance between the exact value and the two closest representable floating-point numbers. The probability is computed as:

$$\circ_{\text{SR}}(x) = \begin{cases} \lfloor x \rfloor & \text{with probability } p_{\text{SR}} \\ \lceil x \rceil & \text{with probability } 1 - p_{\text{SR}} \end{cases} \tag{3}$$

where $p_{\text{SR}}(x)$ is defined as:

$$p_{\text{SR}}(x) = 1 - \frac{x - \lfloor x \rfloor}{\lceil x \rceil - \lfloor x \rfloor}$$

SR implementations include, but are not limited to, the Random Rounding mode of MCA (Parker, 1997) in Verificarlo (Denis et al., 2016), the implementation of Fasi and Mikaitis (Fasi & Mikaitis, 2021) and the average rounding mode of Verrou (Févotte & Lathuiliere, 2016).

2.1.4 Up-Down Rounding

This paper introduces the Up-Down Rounding (UD) technique, which rounds the result of each floating-point operation rounded with the IEEE-754 round-to-nearest rounding mode to the next or previous floating-point number with equal probabilities. Although UD rounding does not preserve exact floating-point operations, it produces results similar to SR rounding on large code bases, while generally being significantly faster. UD rounding is defined as:

$$\circ_{\text{UD}}(x) = \begin{cases} \circ_{\text{RN}}(x) - \epsilon(x) & \text{with probability } \frac{1}{2} \\ \circ_{\text{RN}}(x) + \epsilon(x) & \text{with probability } \frac{1}{2} \end{cases} \tag{4}$$

where $\epsilon(x)$ is the *unit in the last place* (Muller et al., 2018) if $x \neq 0$ and 0 otherwise, and $\circ_{\text{RN}}(x)$ is the IEEE-754 round-to-nearest with ties-to-even rounding mode.

2.2 Floating-Point Error Quantification

We use the significant bits metric (Parker, 1997; Sohier et al., 2021) to quantify the numerical error arising from multiple executions with stochastic arithmetic. This metric represents the amount of shared information among the perturbed results: the more significant bits, the greater the precision. We estimate the number of significant bits using the non-parametric method described in (Sohier et al., 2021), which is implemented in the `significant_digits` package (Verificarlo, 2024).

Given a set of results $X_{i \leq n}$ perturbed by stochastic arithmetic, the number of significant bits $\hat{s_b}$ is defined as:

$$\hat{s_b} = \max \left\{ k \in [\![1, p]\!] \mid \forall i \in [\![1, n]\!],\ S_i^k = \mathbb{1} \right\}$$

where $S_i^k = \mathbb{1}_{Z_i < 2^{-k}}$, and $Z_i = |X_i / x_{ref} - 1|$, with x_{ref} being the reference to compare with (typically the average of $X_{i \leq n}$ or the result computed with the IEEE-754 round-to-nearest mode). Here, p represents the precision (53 bits for IEEE-754 binary64 or 24 bits for binary32 formats).

The k^{th} bit in the significand is considered significant if the absolute value of Z_i is less than 2^{-k}. The number of significant bits is taken as the largest integer k for which this condition holds for all i.

A value of 0 significant bits indicates that X carries no information, while a value of p indicates maximal information, given the floating-point format used. The difference between p and $\hat{s_b}$ quantifies the loss of information resulting from numerical errors.

Significant bits can be converted to significant digits if necessary, such that the maximum number of significant digits for double precision numbers is 15.95 given $53 \log_{10}(2)$ and the maximal number of significant digits for single precision numbers is 7.23 given $24 \log_{10}(2)$.

2.3 Sørensen-Dice Scores

The neuroimaging use case, the FastSurfer CNN, produces brain segmentations from anatomical Magnetic Resonance Images (MRI), which label different brain regions. In order to evaluate brain segmentations, we cannot use the significant bits metric. Segmentation tools like FastSurfer generate categorical variables encoded as integers to represent segmentation labels, even though they rely on floating-point operations. Therefore, the significant bits metric cannot be applied as it is only useful for programs that produce floating-point outputs.

To assess the impact of numerical perturbations on segmentation stability, we compute the minimum Sørensen-Dice score across pairs of SA runs. The Sørensen-Dice score measures the overlap between two segmentation results and is commonly used to quantify similarity between labeled regions in medical imaging. For each subject, we run FastSurfer multiple times with SA-enabled perturbations, producing N different segmentations. Each segmentation output assigns a class label to every voxel in the brain MRI. Let S_i and S_j represent the set of voxels assigned to a specific brain region in the i-th and j-th SA iterations, respectively. The Sørensen-Dice score between these two segmentations is given by:

$$\text{min Sørensen-Dice Score} = \min_{\substack{i,j \in \{1,\dots,N\} \\ i \neq j}} \left(\frac{2 \cdot |S_i \cap S_j|}{|S_i| + |S_j|} \right)$$

where $|S_i \cap S_j|$ is the number of overlapping voxels classified as part of the same region in both segmentations, and $|S_i|$ and $|S_j|$ are the total number of voxels assigned to that region in each iteration.

3 Implementation

This section describes the Fuzzy PyTorch implementation. Subsection 3.1 introduces the **P**robabilistic **R**ounding with **I**nstruction **S**et **M**anagement (PRISM) library, which implements the stochastic rounding and up-down rounding modes. Subsection 3.2 discusses the modifications made to the Verificarlo compiler for compiling PyTorch with the PRISM library. Section 3.3 explains the instrumentation choices made to integrate the PRISM library into PyTorch.

3.1 PRISM

PRISM is a `C++` library that implements the stochastic rounding (sec. 2.1.3) and up-down rounding (sec. 2.1.4) modes. It leverages the Highway library (Google, 2024a) to use vectorized instructions available on modern CPUs, thereby minimizing the overhead introduced by stochastic arithmetic. Highway selects the best architecture target to generate the most efficient code for the CPU, either at compile time (static dispatch) or runtime (dynamic dispatch).

PRISM provides probabilistic rounding (SR and UD) for the floating-point operations $\{+, -, \div, \times, \sqrt{}, \text{FMA}\}$, where FMA refers to Fused Multiply-Add. The SR operators (except FMA) are implemented using the rounding-mode-free algorithms by Fasi & Mikaitis (2021). We extend these algorithms to support the FMA instruction (described in Appendix Algorithm 1). Our FMA implementation is inspired by Verrou and is based on the ErrFmaNearest Algorithm by Boldo & Muller (2010).

PRISM's interface offers functions for scalar and vector instructions, supporting both static and dynamic dispatch. The static dispatch versions accept vector types as inputs, while the dynamic dispatch versions accept pointers to scalar types. Dynamic dispatch is necessary because vector types may not be available at runtime (e.g., 512-bit AVX-512 registers on AVX2 architecture with 256-bit registers). Although slightly slower, dynamic dispatch enhances portability, enabling x86-64 binaries to run on any architecture.

Finally, PRISM supports multithreaded execution by assigning a separate random generator to each thread, ensuring that concurrent executions do not share the same seed state. This enables optimal performance without requiring any synchronization mechanism and prevents correlations in the generated floating-point perturbations across threads.

3.2 Verificarlo

Verificarlo (Denis et al., 2016) is a clang-based compiler that replaces floating-point operations with generic calls to configurable backends (e.g., MCA, IEEE, VPREC (Chatelain et al., 2019)) at the LLVM (Lattner & Adve, 2004) Intermediate Representation (IR) level. In its current version, Verificarlo serializes the vectorized instructions, which can cause additional slowdowns. It adheres to the Interflop (Defour et al., 2021) interface, which exposes scalar arithmetic operations but not vectorized ones. We modified the Verificarlo compiler to use the PRISM library. Specifically, we implemented new LLVM instrumentation passes to replace scalar and vectorized floating-point operations with calls to the PRISM library. We also ensured ABI compatibility between the PRISM library and the source code to prevent incorrect register use during argument passing.

3.3 PyTorch Instrumentation

We instrumented PyTorch version 2.2.1 with Verificarlo 2.0.0, using the PRISM library as the backend. Verificarlo was built with LLVM version 7.0.0, including support for FORTRAN code through LLVM's flang compiler. We used Python 3.8.5. To ensure compatibility with Verificarlo, we modified only one line in the PyTorch codebase. Specifically, we removed the `noexcept` keyword from the move constructor of the `Module` class in `torch/csrc/jit/api/module.h`. This adjustment was necessary to prevent compilation errors related to LLVM compatibility but should no longer be required with more recent LLVM versions.

To achieve complete instrumentation, we compiled the open-source BLAS and LAPACK libraries (Anderson et al., 1999) with Verificarlo, replacing the proprietary Intel MKL library as the default BLAS implementation. Architecture-specific instructions (`-march=native`) were enabled to leverage vectorized operations. Additionally, the ONNX (Community, 2024) runtime was compiled with the PRISM library to ensure comprehensive instrumentation of the entire model execution. We disabled the use of the Intel MKL DNN (Corporation, 2024) library to avoid reliance on proprietary software. We did not instrument the protobuf (Google, 2024b) third-party library to avoid perturbing the serialization of the model. We conducted the experiments with the aforementioned software versions. We have since also instrumented PyTorch version 2.6.0 with Python version 3.10 and LLVM 11.0.0 via Verificarlo 2.2.0.

We excluded specific functions from instrumentation because they were susceptible to producing erroneous outputs under our rounding modes. Correct instrumentation alternatives are under development In particular, PyTorch's exponential and logarithmic functions in the SLEEF (SIMD Library for Evaluating Elementary Functions) third-party library exhibited large output deviations when perturbed, especially near critical input regions such as zero. Similarly, the `torch.argmax` operation became unreliable under up-down rounding, likely because approximate rounding can alter comparisons when input values are close, affecting the selected index. As these functions are sensitive and integral to correct model behavior, they were not instrumented to ensure the reliability of the current results.

4 Experiments

The goal of the experiments in this section is to evaluate the performance and reliability of Fuzzy PyTorch, particularly its ability to handle numerical variability in DL models. We explore the effectiveness of the PRISM library, which implements two rounding modes (stochastic rounding and up-down rounding), and assess their impact on both simple mathematical series and complex DL applications. These experiments aim to compare Fuzzy PyTorch's performance against existing tools such as Verrou, CADNA and Fasi and Mikaitis' stochastic rounding library. Additionally, we confirm Fuzzy PyTorch's scalability by successfully running it on models ranging from 1 to 341 million parameters, demonstrating its applicability across small-scale and large-scale deep learning architectures.

4.1 Numerical Error Estimation Tools

We compared the PRISM library with state-of-the-art tools for numerical variability analysis, as detailed in this section.

4.1.1 Verrou

Verrou is a Valgrind-based tool (Nethercote & Seward, 2007) used to evaluate programs with stochastic arithmetic. Verrou uses the Dynamic Binary Instrumentation (DBI) technique to replace floating-point operations with calls to its own rounding functions in the Valgrind virtual machine. It offers several modes to perturb floating-point operations, including SR (referred to as average rounding in Verrou) and asynchronous CESTAC (referred to as random rounding in Verrou). Since Verrou is based on Valgrind, it can instrument any program without the need for recompilation or access to the source code. Verrou applies stochastic rounding to each arithmetic floating-point operation in the set $\{+, -, \div, \times, \sqrt{}, \text{FMA}\}$.

Verrou minimizes overhead by efficiently implementing rounding operations, random number generation, and using FMA instructions when possible. However, Valgrind serializes multi-threaded executions, which can lead to further slowdowns.

4.1.2 MCA Backend

The Monte Carlo Arithmetic (Parker, 1997) (MCA) backend of Verificarlo perturbs floating-point operations by introducing random values drawn from a uniform distribution. Users can control the magnitude of these perturbations using the concept of virtual precision (t). To avoid double rounding errors, the backend performs intermediate computations at a precision twice that of the target precision (p). This backend supports various rounding modes, including Random Rounding (RR), which is effectively equivalent[1] to Stochastic Rounding (SR) when $t = p$.

While the MCA backend is thread-safe, it serializes vectorized instructions, which can lead to performance slowdowns. Additionally, when perturbing binary64 numbers, it relies on 128-bit arithmetic (either binary128 or an optimized 128-bit integer implementation), which introduces extra computational overhead.

4.1.3 CADNA

CADNA (Jézéquel & Chesneaux, 2008) is a library implementing the CESTAC technique, a synchronous stochastic arithmetic method. In CADNA, perturbations are applied to three floating-point values during each operation. The library tracks these perturbed values and calculates the number of significant digits when performing floating-point comparisons or generating string representations. This enables CADNA to precisely identify when floating-point values lose all significance with the notion of "computational zero".

CADNA is provided as a `C++` library, offering stochastic types (`float_st`, `double_st`) that can replace standard floating-point types in a codebase. Through operator overloading, the library allows for the transparent execution of existing code. Additionally, CADNA includes the `cadnaizer` tool, which automates the replacement of floating-point types within a codebase.

For parallel computing, CADNA offers a thread-safe version built on the OpenMP library. However, one key limitation of the library is its lack of support for vectorized instructions.

4.1.4 Fasi and Mikaitis stochastic rounding

Fasi & Mikaitis (2021) developed an implementation of their algorithms in the form of a `C` library. This library provides stochastic rounding for scalar floating-point arithmetic operations, except for the FMA instruction. Each operation must be replaced by a call to the corresponding function in the library. The library does not offer an interface for vectorized instructions. We will refer to this library as FM SR in the remainder of the paper.

[1]The MCA RR mode is biased for inputs close to a power of two, see (de Oliveira Castro, 2022).

4.2 Use cases

We evaluated the accuracy and performance of Fuzzy PyTorch across multiple use cases: (1) the harmonic series to compare numerical behaviors of stochastic rounding (SR) and up-down rounding (UD) modes against state-of-the-art numerical tools; (2) the NAS Parallel Benchmarks, widely used for performance assessment in high-performance computing; and (3) three deep-learning models demonstrating real-world applicability —MNIST (handwritten digit classification), WavLM (applied to speech-based Parkinson's disease identification), and FastSurfer (brain segmentation using CNN).

4.2.1 Harmonic Series

To evaluate the accuracy of the UD and SR rounding modes, we analyzed the harmonic series $\sum_{i=1}^{n} \frac{1}{i}$ for n ranging from 10^2 to 10^7. While this series is divergent in real arithmetic, it converges in floating-point arithmetic due to numerical absorption. Notably, the series is known to converge for $n \geq 2^{48}$ in the binary32 format (Malone, 2013). The code was compiled using the IEEE-754 binary32 format. For comparison, we computed a reference value using the binary64 format. We evaluated the PRISM SR and UD rounding modes against CESTAC (via the CADNA library), MCA RR (using Verificarlo), Verrou SR, Verrou CESTAC, and FM SR.

4.2.2 NAS Parallel Benchmarks

To evaluate the performance of the PRISM library, we compared it against existing numerical analysis tools using the `C++` NAS Parallel Benchmarks 3.4.1 by Löff et al. (2021) (cf. Appendix Table 1). The IS benchmark was excluded from our analysis due to its lack of floating-point operations. We conducted experiments on datasets of class S, testing the serial implementation (SER). We assessed the PRISM SR and UD rounding modes against several numerical analysis tools, including CESTAC (via the CADNA library), MCA RR (using Verificarlo), Verrou SR, Verrou CESTAC, and FM SR. We average performance measurements across 3 repetitions for each configuration. All benchmarks were compiled with `-march=haswell -maes` to enable AVX2 instructions. Computations were performed using the IEEE-754 binary64 format. Additionally, we used a version without instrumentation (IEEE) as a baseline for comparison.

4.2.3 MNIST

To evaluate the performance and behavior of Fuzzy PyTorch, we conducted experiments on a small CNN trained on the MNIST dataset. The model architecture consisted of convolutional, ReLU, max-pooling, convolutional, ReLU, dropout, flatten, linear, ReLU, dropout, and linear layers, followed by a log-softmax output layer. The task, a classification problem to identify digits from the input images, is well-established and widely considered solved. This experiment served as a baseline to demonstrate the feasibility and potential benefits of Fuzzy PyTorch in a controlled, well-understood context.

4.2.4 WavLM

The model for Parkinson's identification from speech is a pipeline composed of the WavLM Large (Chen et al., 2022) model in a frozen configuration to extract features from the audio recordings that are fed to the Emphasized Channel Attention, Propagation, and Aggregation Time Delay Neural Network (ECAPA-TDNN) (Desplanques et al., 2020) and a linear layer classifier followed by a max operation to obtain binary classification to determine whether a subject has Parkinson's disease or is a healthy control subject. The model was trained on the mPower speech dataset (Bot et al., 2016). We will refer to this pipeline as WavLM in this work.

4.2.5 FastSurfer

FastSurfer (Henschel et al., 2020) is a CNN model that performs whole-brain segmentation, cortical surface reconstruction, fast spherical mapping, and cortical thickness analysis from anatomical MRI. The FastSurfer CNN is inspired by the QuickNAT model (Roy et al., 2019), which is composed of three 2D fully convolutional neural networks—each associated with a different 2D slice orientation—that each have the same

encoder/decoder U-net architecture with skip connections, unpooling layers and dense connections as QuickNAT. A diagram of the model's architecture is available in the Appendix Figure 8. We focus exclusively on the task of whole-brain segmentation, defined as voxel-wise anatomical labeling of brain regions. This segmentation step is entirely performed by the CNN, without surface reconstruction, and uses the pre-trained FastSurfer model (v2.1.0) available on GitHub (Deep-MI, 2024). FastSurfer has demonstrated high accuracy, strong generalization to unseen datasets, and high test-retest reliability.

This model serves as an ideal benchmark for studying numerical variability in high-dimensional medical imaging tasks due to its scientific relevance and architectural complexity. In our experiments, we applied FastSurfer to segment five subjects from the CoRR dataset (Zuo et al., 2014). Standard inference By comparing Fuzzy PyTorch and Verrou instrumentation on FastSurfer inference, we aim to partially replicate previous findings on numerical uncertainty in FastSurfer (Gonzalez-Pepe et al., 2023), thereby validating the accuracy, reliability, and applicability of Fuzzy PyTorch to realistic large-scale segmentation tasks.

4.3 Availability of Data and Code

The code and data used in this study are available on GitHub at `https://github.com/big-data-lab-team/fuzzy-pytorch`. The repository includes the PRISM library, the modified Verificarlo compiler, the Dockerfile to build Fuzzy PyTorch, and the scripts used to run the experiments. Additionally, the repository contains the NAS Parallel Benchmarks, the FastSurfer model, and the MNIST dataset.

4.4 Computational Infrastructure

All experiments except for the WavLM experiment were conducted on a server equipped with 8 compute nodes with 32 cores Intel(R) Xeon(R) Gold 6130 CPU @ 2.10GHz 22MB cache L3. The WavLM experiment was conducted on the Narval cluster from École de Technologie Supérieure (ETS, Montréal), managed by Calcul Québec and The Digital Alliance of Canada which includes AMD Rome 7502, AMD Rome 7532, and AMD Milan 7413 CPUs with 48 to 64 physical cores, 249 GB to 4000 GB of RAM and Linux kernel 3.10.

5 Results

In this section, we present detailed experimental results evaluating the accuracy, runtime efficiency, and numerical variability of Fuzzy PyTorch. First, we perform a sanity check using the harmonic series to compare stochastic rounding (SR) and up-down rounding (UD) modes against state-of-the-art numerical analysis tools. Next, we assess computational overhead with the NAS Parallel Benchmarks, demonstrating Fuzzy PyTorch's performance in an HPC context. Finally, we measure runtime performance and numerical variability in three practical deep-learning applications: MNIST digit classification, FastSurfer brain segmentation, and WavLM-based Parkinson's disease identification.

5.1 Harmonic Series Validates Expected Variability Patterns

In Figure 1, the analysis of the harmonic series $\sum_{i=1}^{n} \frac{1}{i}$ for n ranging from 10^2 to 10^7 included the standard deviation and mean values obtained from three repetitions for each mode. For CADNA, a single run was performed, and the three sample values were extracted from its stochastic type. In Figure 1, the results show that the PRISM SR mode exhibits variability comparable to MCA RR, Verrou SR, and FM SR (Levene's test, $p = 0.29$, Extended Data Table 2). In contrast, UD rounding displays higher variability, as anticipated, due to approximating function outputs. CADNA, due to its synchronous nature, exhibits even higher variability, as it ensures that its three repetitions are rounded in different directions.

As shown in Figure 1, the mean values revealed three distinct behaviors:

1. The MCA SR, Verrou SR, PRISM SR, and FM SR modes closely approximate the binary64 reference value. This aligns with expectations, as SR rounding converges to the expected value.
2. The CESTAC and Verrou CESTAC modes diverge more rapidly. This behavior is consistent with the known bias introduced by CESTAC rounding.

3. The PRISM UD mode converges toward the binary32 result, as expected, since UD rounding applies random perturbations to values that have already been rounded using the round-to-nearest mode.

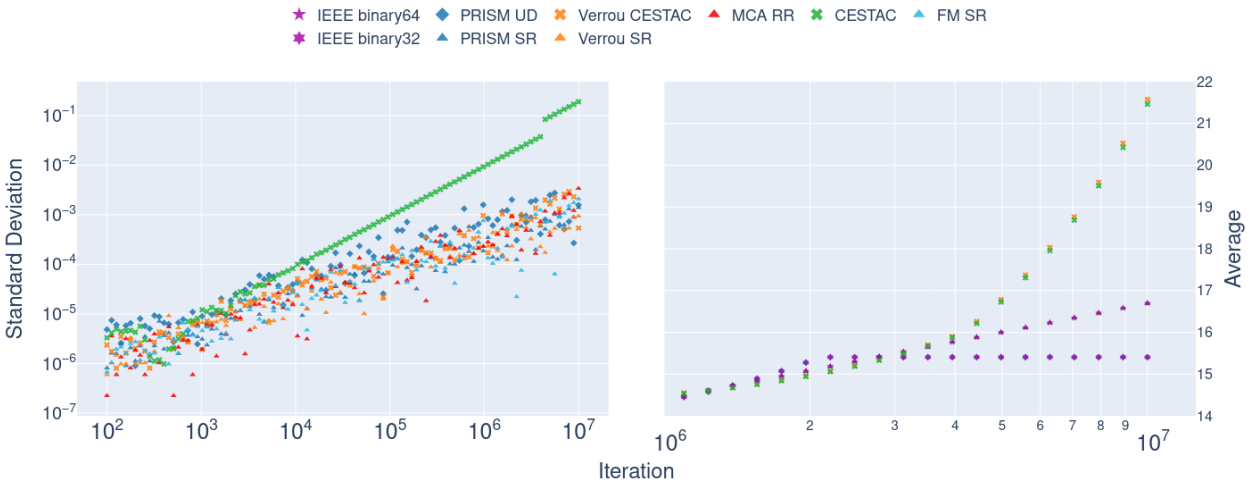


Figure 1: Comparison of probabilistic rounding on the harmonic series example.

5.2 PRISM UD Minimizes Runtime Slowdowns for NAS Parallel Benchmarks

The runtime analysis of the NAS Parallel Benchmarks (NPB) on dataset S highlights the slowdowns introduced by various numerical variability analysis tools. The results, shown in Figure 2, demonstrate that the PRISM SR mode exhibited runtime slowdowns comparable to Verrou SR and CADNA. Notably, the PRISM UD mode consistently outperformed all other tools, achieving the lowest slowdowns across the benchmarks.

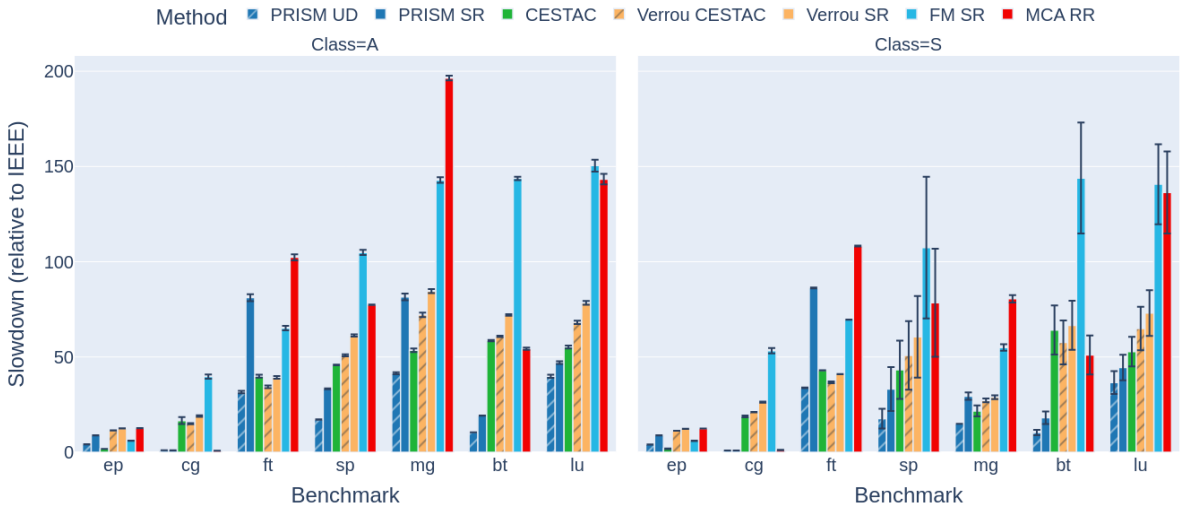


Figure 2: Comparison of slowdowns across numerical variability analysis tools for NAS Parallel Benchmarks on dataset S and A, relative to the IEEE binary64 baseline.

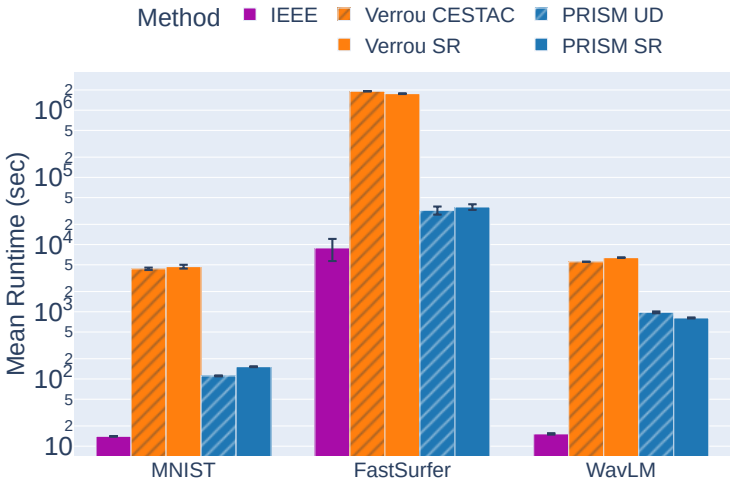


Figure 3: Comparison of runtime across instrumentation tools and use cases in seconds

5.3 Fuzzy PyTorch Achieves Significant Runtime Speedup

To further evaluate the performance of Fuzzy PyTorch, we measured the runtime for three DL use cases: MNIST, FastSurfer and WavLM. Both UD and SR modes were tested and compared respectively to the CESTAC and SR modes in Verrou. All experiments were executed in single-threaded mode, as Verrou sequentializes multithreaded codes.

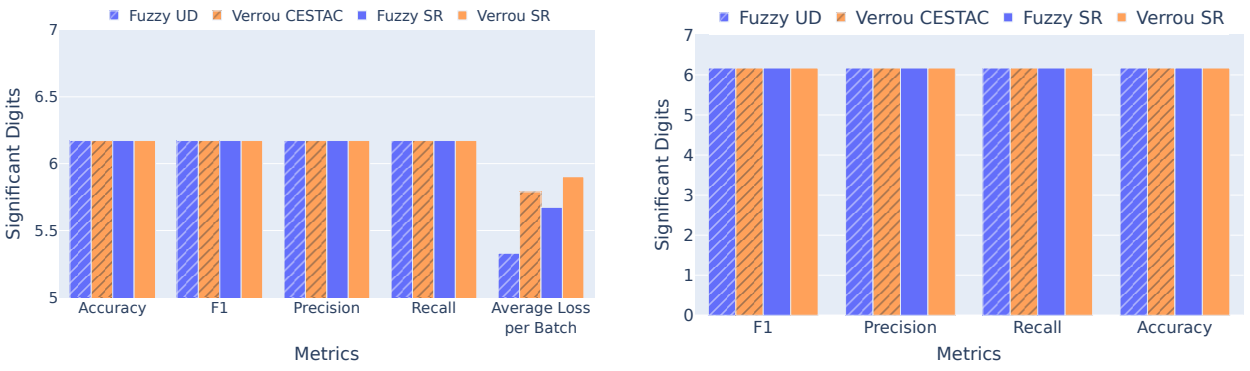
As shown in Figure 3, Fuzzy PyTorch achieves speedups of 5× to 60× for UD mode and 7× to 49× for SR mode compared to Verrou's implementations. This improvement is notable, especially considering that Verrou's version of PyTorch relies on Intel MKL—a highly optimized mathematical library—while Fuzzy PyTorch uses OpenBLAS and LAPACK, which are less optimized.

In the NAS parallel benchmarks, PRISM UD was generally faster than PRISM SR, consistent with its approximation mechanism. In our deep learning use cases, however, we find that while both modes achieve substantial speedups over Verrou, UD is not always the faster approach. PRISM UD is slower than PRISM SR for WavLM, with speedups of 5.65× for UD and 7.89× for SR relative to Verrou. In purely CNN architectures, such as FastSurfer (UD: 60.43×, SR: 49.07×) and MNIST (UD: 39.22×, SR: 30.81×), UD maintains its advantage. We attribute the WavLM slowdown in UD mode to the fact that the WavLM model contains a transformer model, which is computationally more complex than CNNs: self-attention layers introduce quadratic cost in sequence length and heavier intermediate memory usage. Since SR preserves exact operations, it avoids extra perturbation computations in these components, thereby reducing its relative overhead, whereas UD's cost remains largely constant regardless of the underlying operation type.

5.4 Comparable Variability between Fuzzy PyTorch and Verrou

To assess Fuzzy PyTorch beyond runtime performance, we evaluate the variability it introduces compared to Verrou. This evaluation is conducted on three use cases: MNIST digit classification, FastSurfer brain segmentation and WavLM Parkinson's classification.

For the MNIST use case, we examine standard machine learning metrics, including accuracy, weighted variations of precision, recall, and F1 score. In binary32, the theoretical maximum precision is 7.23 significant digits; here, we observed a maximum of about 6.17 significant digits across all metrics. Exception made for the loss function that showed higher variability and thus fewer significant digits. This behavior is expected because MNIST classification is a well-solved problem with high prediction stability, leaving little room for significant variability.



(a) MNIST (b) WavLM

Figure 4: Significant digits across MNIST and WavLM model metrics for different instrumentation tools.

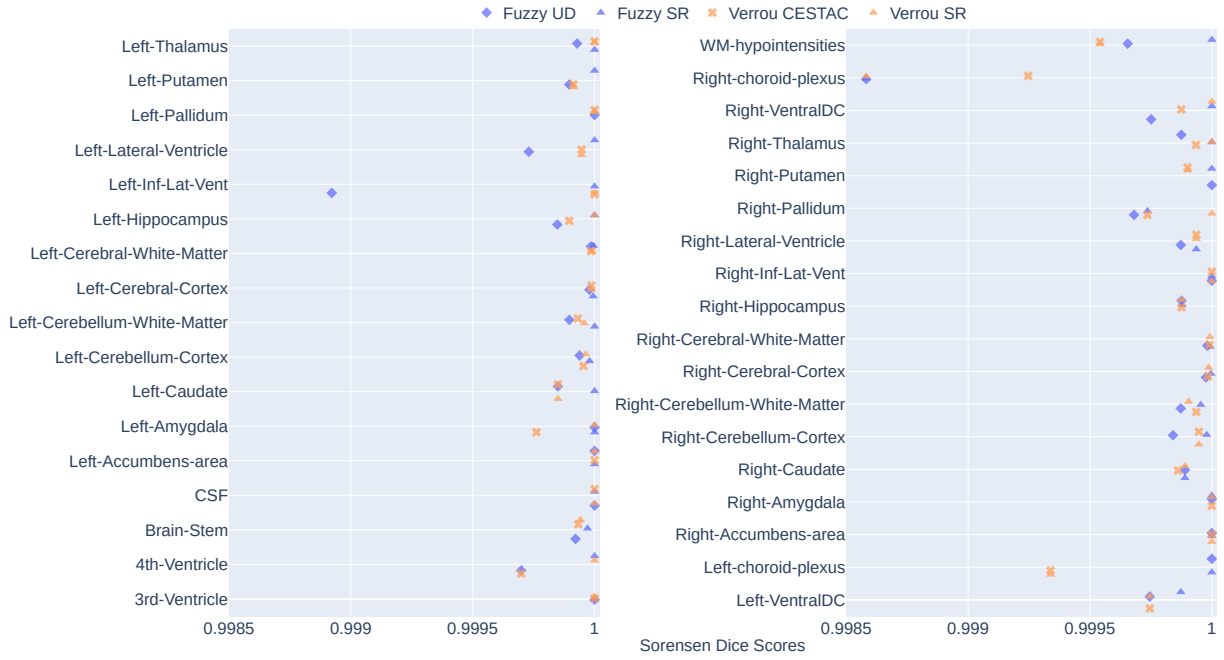


Figure 5: Minimum Sørensen-Dice score across instrumentation tools and different labelled brain regions, denoted as Regions of Interest (ROI).

As shown in Figure 4a, we only observe variability in the loss function across the Fuzzy PyTorch and Verrou tools as expected. We note that UD mode consistently introduces greater variability than stochastic rounding for both tools. Nonetheless, Fuzzy PyTorch exhibits slightly larger variability overall, as indicated by its lower significant digits. We attribute this difference to the instrumentation of AVX-512 instructions in Fuzzy PyTorch, which allows a broader range of floating-point operations to be perturbed compared to Verrou.

For the FastSurfer use case, we assess variability at inference using the minimum Sørensen–Dice scores between MCA iterations (Figure 5). The minimum Sørensen–Dice score captures the most extreme cases of variability across brain regions, offering a global measure of segmentation consistency. Across all modes, coefficients remain extremely high, with the lowest observed value approaching 0.9985. Variability magnitudes are similar across methods, though slightly higher for PRISM UD—likely due to its lack of preservation of exact operations. Similarly to the MNIST results, PRISM UD shows the highest variability but still maintains comparable segmentation accuracy. These results confirm that our MCA-based instrumentation for FastSurfer operates correctly, producing consistent and interpretable variability measurements across modes.

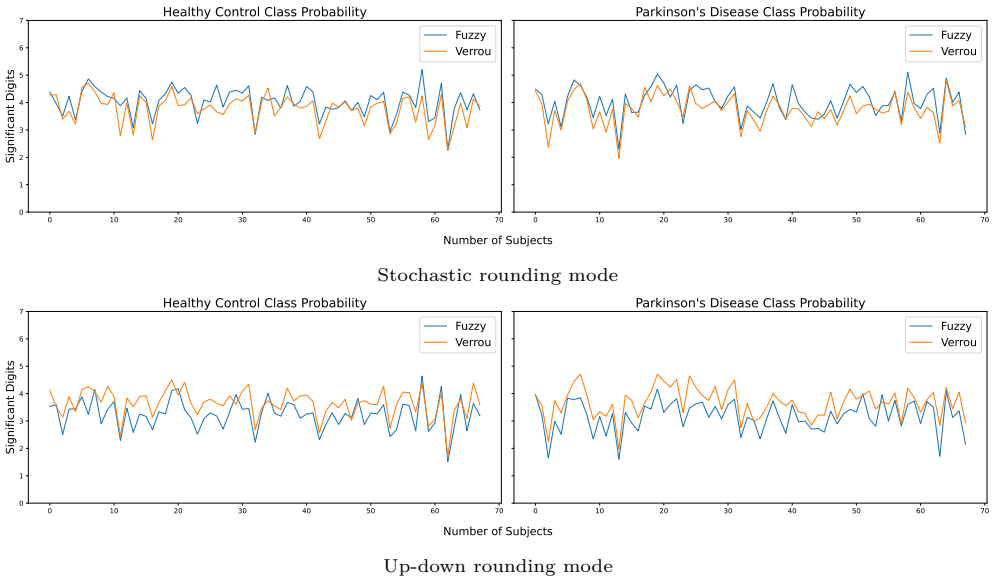


Figure 6: Significant digits for rounding modes across WavLM model's class probabilities

In the WavLM use case, as with MNIST, we evaluated numerical variability across accuracy and the weighted variants of precision, recall, and F1 score. On average, we observe 6.17 significant digits across all performance metrics, indicating high numerical stability—comparable to that observed across IEEE executions.

To investigate whether this apparent stability conceals underlying instability, we analyzed the model's output probabilities before the final max operation. As shown in Figure 6, the number of significant digits drops, averaging around 4 across all modes and tools, with a standard deviation of approximately half a digit. This suggests that some numerical instability is indeed present but is masked by the final max operation. Consistent with findings from previous use cases, we also note that PRISM UD mode introduces the highest level of numerical perturbation—even surpassing Verrou's CESTAC mode.

For all use cases, we verified that no random processes were present after fixing the random seeds by running multiple iterations of the IEEE implementations of each model. In FastSurfer, this yielded a perfect minimum Sørensen–Dice score (1.0, standard deviation 0), confirming determinism. For MNIST and WavLM, all metrics, including loss, retained 6.17 significant digits.

5.5 Model Embeddings Are Comparable

Furthermore, we analyzed the internal layers of the models, referred to as embeddings, for all use cases to ensure that numerical perturbations are not limited to the outputs but also occur consistently across intermediate stages. Figure 7 illustrates one layer from each use case: the output of the second decoder block for FastSurfer, the first convolutional layer for MNIST and the output of the ECAPA-TDNN component of the WavLM model. The numerical variability patterns were consistent across all cases, though, for WavLM, we can visibly see the higher numerical uncertainty across all tools, especially with Fuzzy PyTorch's up-down rounding mode.

While the numerical uncertainty within the MNIST and WavLM embeddings aligns with the observed stability of its outputs, FastSurfer presented a notable discrepancy: substantial uncertainty in the background of its embeddings despite its stable outputs.

Further investigation uncovered that this instability was confined to the background region outside the brain and originated from unstable indices being generated by the max-pooling operation across SA iterations. Post-processing steps in FastSurfer effectively masked the background, thereby mitigating the impact of these instabilities on the final outputs for this specific use case. This finding directly motivated the development

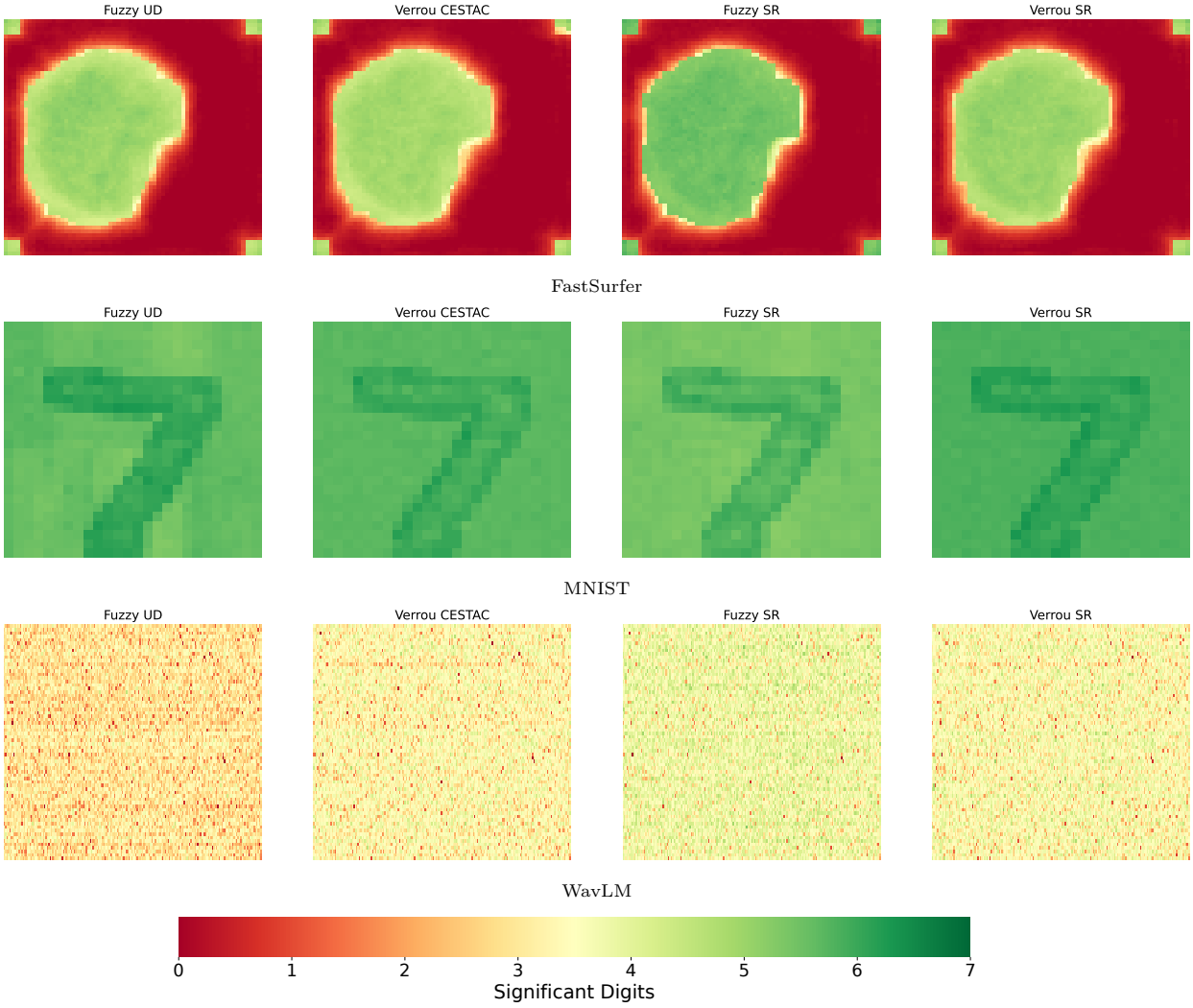


Figure 7: Model embeddings across stochastic arithmetic implementations

of Conservative & Aggressive NaNs, two approaches for leveraging numerical uncertainty into computational efficiency gains while preserving model performance (Anonymized, c).

Overall, these findings demonstrate consistency between the global patterns observed in Fuzzy PyTorch and Verrou's results, reinforcing the reliability of Fuzzy PyTorch in assessing variability throughout DL models.

6 Conclusion

Fuzzy PyTorch is a framework designed to evaluate numerical variability in DL models, addressing the challenges posed by floating-point arithmetic limitations. By leveraging vectorized CPU instructions via the Highway library, it minimizes computational overhead while maintaining flexibility. The framework supports two rounding modes, SR and UD, allowing researchers to balance precision and computational efficiency, making it a versatile tool for enhancing model robustness and reproducibility.

Our analysis of the harmonic series reveals distinct behaviors among rounding modes: SR closely matches the binary64 reference, demonstrating high precision; UD tends toward the binary32 result due to random perturbations; and CESTAC diverges rapidly because of its inherent bias. These patterns illustrate the subtle differences in numerical variability introduced by various rounding strategies. These same patterns emerged in our deep learning experiments, with SR consistently delivering the most accurate results across tasks. Crucially, Fuzzy PyTorch produces numerical variability measurements comparable to those of state-of-the-art frameworks such as Verrou, as demonstrated across MNIST classification, FastSurfer segmentation, and WavLM Parkinson's detection tasks.

Moreover, Fuzzy PyTorch achieves substantial runtime speedups. In the NAS parallel benchmarks we see minimal slowdowns for PRISM SR and especially PRISM UD. In the deep learning tasks, we see up to 60× speedup depending on the use case—despite relying on less optimized CPU libraries (OpenBLAS and LAPACK versus Intel MKL). This efficiency gain enables the scalable evaluation of numerical variability in large deep learning models, a capability unmatched by existing methods.

While our current implementation is CPU-based, the main conclusions from CPU testing are expected to generalize to GPU architectures. Fuzzy PyTorch offers a scalable framework for evaluating numerical variability in DNNs—no existing method achieves this at scale. Although CPU execution is a limitation, extending the framework to GPU architectures poses significant technical challenges. The GPU tooling ecosystem is far more fragmented than its CPU counterpart, requiring specialized approaches for different vendor platforms. A successful GPU implementation would need to address multiple compilation targets and runtime environments. For NVIDIA GPUs, dynamic instrumentation frameworks like NVBit (Villa et al., 2019) could enable analysis of closed-source libraries, but this approach is complex and platform-specific. Alternatively, compiler-based solutions using IREE (Liu et al., 2022) with specific compilation targets could offer broader hardware support, though this would require substantial development effort to integrate with existing PyTorch workflows. The framework will be applicable to GPU-based deep learning workloads once Verificarlo extends its support to GPU architectures.

Future directions for Fuzzy PyTorch include exploring the impact of numerical variability on diverse DL model architectures, optimizing SR mode performance, extending PRISM with additional floating-point formats and specialized DL instructions, and investigating solutions for the instabilities uncovered by Fuzzy PyTorch within FastSurfer's internal layers.

In summary, Fuzzy PyTorch provides an efficient, reliable, and versatile tool for assessing numerical variability in DL models. It empowers researchers to deepen their understanding of numerical behavior in DL models and enhances their ability to develop robust and reproducible systems.

As numerical variability becomes increasingly recognized across the AI industry, the availability of system-level controls—such as AWS Neuron's hardware-supported rounding modes—highlights the pressing need for software frameworks like Fuzzy PyTorch. These frameworks make it possible to perform fine-grained evaluations of numerical behavior across diverse architectures, from CPUs to accelerators, and ensure that both academic and industrial workflows benefit from deeper guarantees of computational robustness.

References

Edward Anderson, Zhaojun Bai, Christian Bischof, L Susan Blackford, James Demmel, Jack Dongarra, Jeremy Du Croz, Anne Greenbaum, Sven Hammarling, Alan McKenney, et al. *LAPACK users' guide*. SIAM, 1999.

Anonymized. a.

Anonymized. b.

Anonymized. c.

Anonymized. d.

Anonymized. e.

Anonymized. f.

El-Mehdi El Arar, Silviu-Ioan Filip, Theo Mary, and Elisa Riccietti. Mixed precision accumulation for neural network inference guided by componentwise forward error analysis. *arXiv preprint arXiv:2503.15568*, 2025.

AWS Neuron Team. Rounding modes. `https://awsdocs-neuron.readthedocs-hosted.com/en/latest/general/arch/neuron-features/rounding-modes.html`, n.d. Accessed: 2025-06-25.

Théo Beuzeville, Alfredo Buttari, Serge Gratton, and Theo Mary. Deterministic and probabilistic backward error analysis of neural networks in floating-point arithmetic. 2024.

Sylvie Boldo and Guillaume Melquiond. Flocq: A unified library for proving floating-point algorithms in coq. In *2011 IEEE 20th Symposium on Computer Arithmetic*, pp. 243–252. IEEE, 2011.

Sylvie Boldo and Jean-Michel Muller. Exact and approximated error of the fma. *IEEE Transactions on Computers*, 60(2):157–164, 2010.

Brian M Bot, Christine Suver, Elias Chaibub Neto, Michael Kellen, Arno Klein, Christopher Bare, Megan Doerr, Abhishek Pratap, John Wilbanks, E Dorsey, et al. The mpower study, parkinson disease mobile data collected using researchkit. *Scientific data*, 3(1):1–9, 2016.

Marie-Christine Brunet and Françoise Chatelin. Cestac, a tool for a stochastic round-off error analysis in scientific computing. In *Numerical Mathematics and Applications*, pp. 11–20. Elsevier, 1986.

Yohan Chatelain, Eric Petit, Pablo de Oliveira Castro, Ghislain Lartigue, and David Defour. Automatic exploration of reduced floating-point representations in iterative methods. In *Euro-Par 2019: Parallel Processing: 25th International Conference on Parallel and Distributed Computing, Göttingen, Germany, August 26–30, 2019, Proceedings 25*, pp. 481–494. Springer, 2019.

Sanyuan Chen, Chengyi Wang, Zhengyang Chen, Yu Wu, Shujie Liu, Zhuo Chen, Jinyu Li, Naoyuki Kanda, Takuya Yoshioka, Xiong Xiao, et al. Wavlm: Large-scale self-supervised pre-training for full stack speech processing. *IEEE Journal of Selected Topics in Signal Processing*, 16(6):1505–1518, 2022.

ONNX Community. Onnx: Open neural network exchange, 2024. URL `https://github.com/onnx/onnx`. Accessed: 2024-12-21.

Intel Corporation. onednn: Deep learning open source performance library, 2024. URL `https://github.com/oneapi-src/oneDNN`. Accessed: 2024-12-21.

Pablo de Oliveira Castro. *High Performance Computing code optimizations: Tuning performance and accuracy*. PhD thesis, Université Paris-Saclay, 2022.

Deep-MI. Fastsurfer, 2024. URL `https://github.com/Deep-MI/FastSurfer`. Accessed: 2024-12-17.

David Defour, François Févotte, Stef Graillat, Fabienne Jézéquel, Wilfried Kirschenmann, Jean-Luc Lamotte, Bruno Lathuilière, Yves Lhuillier, Eric Petit, Julien Signoles, et al. Interflop, interoperable tools for computing, debugging, validation and optimization of floating-point programs. In *ISC-HPC 2021 DIGITAL*, 2021.

Christophe Denis, Pablo de Oliveira Castro, and Eric Petit. Verificarlo: checking floating point accuracy through monte carlo arithmetic. In *2016 IEEE 23nd Symposium on Computer Arithmetic (ARITH)*, 2016.

Brecht Desplanques, Jenthe Thienpondt, and Kris Demuynck. Ecapa-tdnn: Emphasized channel attention, propagation and aggregation in tdnn based speaker verification. *arXiv preprint arXiv:2005.07143*, 2020.

El-Mehdi El Arar. Probabilistic error analysis of limited-precision stochastic rounding. `https://github.com/riakymch/pasc25-ms2a/blob/main/pasc25-elarar.pdf`, 2025. Accessed: June 25, 2025.

Julian Faraone and Philip Leong. Monte Carlo Deep Neural Network Arithmetic. 2019.

Massimiliano Fasi and Mantas Mikaitis. Algorithms for stochastically rounded elementary arithmetic operations in ieee 754 floating-point arithmetic. *IEEE Transactions on Emerging Topics in Computing*, 9(3): 1451–1466, 2021.

François Févotte and Bruno Lathuiliere. Verrou: a cestac evaluation without recompilation. *SCAN 2016*, pp. 47, 2016.

George E Forsythe. Reprint of a note on rounding-off errors. *SIAM review*, 1(1):66, 1959.

Inés Gonzalez-Pepe, Vinuyan Sivakolunthu, Hae Lang Park, Yohan Chatelain, and Tristan Glatard. Numerical uncertainty of convolutional neural networks inference for structural brain mri analysis. In *International Workshop on Uncertainty for Safe Utilization of Machine Learning in Medical Imaging*, pp. 64–73. Springer, 2023.

Google. Highway, 2024a. URL `https://github.com/google/highway`. Accessed: 2024-12-17.

Google. Protocol buffers, 2024b. URL `https://github.com/protocolbuffers/protobuf`. Accessed: 2024-12-21.

Leonie Henschel, Sailesh Conjeti, Santiago Estrada, Kersten Diers, Bruce Fischl, and Martin Reuter. Fastsurfer-a fast and accurate deep learning based neuroimaging pipeline. *NeuroImage*, 219:117012, 2020.

Timothy Hickey, Qun Ju, and Maarten H Van Emden. Interval arithmetic: From principles to implementation. *Journal of the ACM (JACM)*, 48(5):1038–1068, 2001.

IEEE Computer Society. IEEE Standard for Floating-Point Arithmetic. `https://ieeexplore.ieee.org/document/4610935`, aug 2008. IEEE Std 754-2008.

Fabienne Jézéquel and Jean-Marie Chesneaux. Cadna: a library for estimating round-off error propagation. *Computer Physics Communications*, 178(12):933–955, 2008.

Eliska Kloberdanz, Kyle G Kloberdanz, and Wei Le. DeepStability: A Study of Unstable Numerical Methods and Their Solutions in Deep Learning. *arXiv preprint arXiv:2202.03493*, 2022.

Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. CGO 2004.*, pp. 75–86. IEEE, 2004.

Hsin-I Cindy Liu, Marius Brehler, Mahesh Ravishankar, Nicolas Vasilache, Ben Vanik, and Stella Laurenzo. Tinyiree: An ml execution environment for embedded systems from compilation to deployment. *IEEE micro*, 42(5):9–16, 2022.

Júnior Löff, Dalvan Griebler, Gabriele Mencagli, Gabriell Araujo, Massimo Torquati, Marco Danelutto, and Luiz Gustavo Fernandes. The nas parallel benchmarks for evaluating c++ parallel programming frameworks on shared-memory architectures. *Future Generation Computer Systems*, 125:743–757, 2021.

David Malone. To what does the harmonic series converge? *Irish Mathematical Society Bulletin*, (71):59–66, 2013.

Jean-Michel Muller, Nicolas Brisebarre, Florent De Dinechin, Claude-Pierre Jeannerod, Vincent Lefevre, Guillaume Melquiond, Nathalie Revol, Damien Stehlé, Serge Torres, et al. *Handbook of floating-point arithmetic*, volume 1. Springer, 2018.

Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan notices*, 42(6):89–100, 2007.

Douglass Stott Parker. *Monte Carlo arithmetic: exploiting randomness in floating-point arithmetic*. Citeseer, 1997.

Abhijit Guha Roy, Sailesh Conjeti, Nassir Navab, Christian Wachinger, Alzheimer's Disease Neuroimaging Initiative, et al. Quicknat: A fully convolutional network for quick and accurate segmentation of neuroanatomy. *NeuroImage*, 186:713–727, 2019.

Devan Sohier, Pablo De Oliveira Castro, François Févotte, Bruno Lathuilière, Eric Petit, and Olivier Jamond. Confidence intervals for stochastic arithmetic. *ACM Transactions on Mathematical Software (TOMS)*, 47 (2):1–33, 2021.

Alexey Solovyev, Marek S Baranowski, Ian Briggs, Charles Jacobsen, Zvonimir Rakamarić, and Ganesh Gopalakrishnan. Rigorous estimation of floating-point round-off errors with symbolic taylor expansions. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 41(1):1–39, 2018.

Verificarlo. significantdigits, 2024. URL `https://github.com/verificarlo/significantdigits`. Accessed: 2024-12-17.

Oreste Villa, Mark Stephenson, David Nellans, and Stephen W Keckler. Nvbit: A dynamic binary instrumentation framework for nvidia gpus. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 372–383, 2019.

Xi-Nian Zuo, Jeffrey S Anderson, Pierre Bellec, Rasmus M Birn, Bharat B Biswal, Janusch Blautzik, John Breitner, Randy L Buckner, Vince D Calhoun, F Xavier Castellanos, et al. An Open Science Resource for Establishing Reliability and Reproducibility in Functional Connectomics. *Scientific Data*, 1(1):1–13, 2014.

A

A.1 Probabilistic Rounding Algorithms

Algorithm 1: FMA With Stochastic Rounding Without the Change of the Rounding Mode

function FMA2($a \in \mathcal{F}$, $b \in \mathcal{F}$, $c \in \mathcal{F}$)
  Compute $\varrho = \circ_{\mathrm{SR}}(a \cdot b + c) \in \mathcal{F}$
  $Z \leftarrow \mathrm{rand}()$
  $\sigma \leftarrow \circ_{\mathrm{RN}}(a \cdot b + c)$
  $(u_1, u_2) \leftarrow \mathrm{TwoProdFMA}(a, b)$
  $(\alpha_1, \alpha_2) \leftarrow \mathrm{TwoSum}(c, u_2)$
  $(\beta_1, \beta_2) \leftarrow \mathrm{TwoSum}(u_1, \alpha_1)$
  $\gamma \leftarrow \circ_{\mathrm{RN}}(\circ_{\mathrm{RN}}(\beta_1 - r_1) - \beta_2)$
  $\tau \leftarrow \circ_{\mathrm{RN}}(\gamma + \alpha_2)$
  $\mathrm{round} \leftarrow \mathrm{SRround}(\sigma, \tau, Z)$
  $\varrho \leftarrow \circ_{\mathrm{RN}}(r_1 + \mathrm{round})$
  Return result
end function

A.1.1 NAS Parallel Benchmarks description

Benchmark	Description
bt	Block Triangular Solver
cg	Conjugate Gradient
ep	Embarrassingly Parallel
ft	Fast Fourier Transform
lu	Lower-Upper Symmetric Gauss-Seidel
mg	Multi-Grid Solver
sp	Scalar Pentadiagonal Solver

Table 1: NAS Parallel Benchmarks description

A.2 Model Architecture

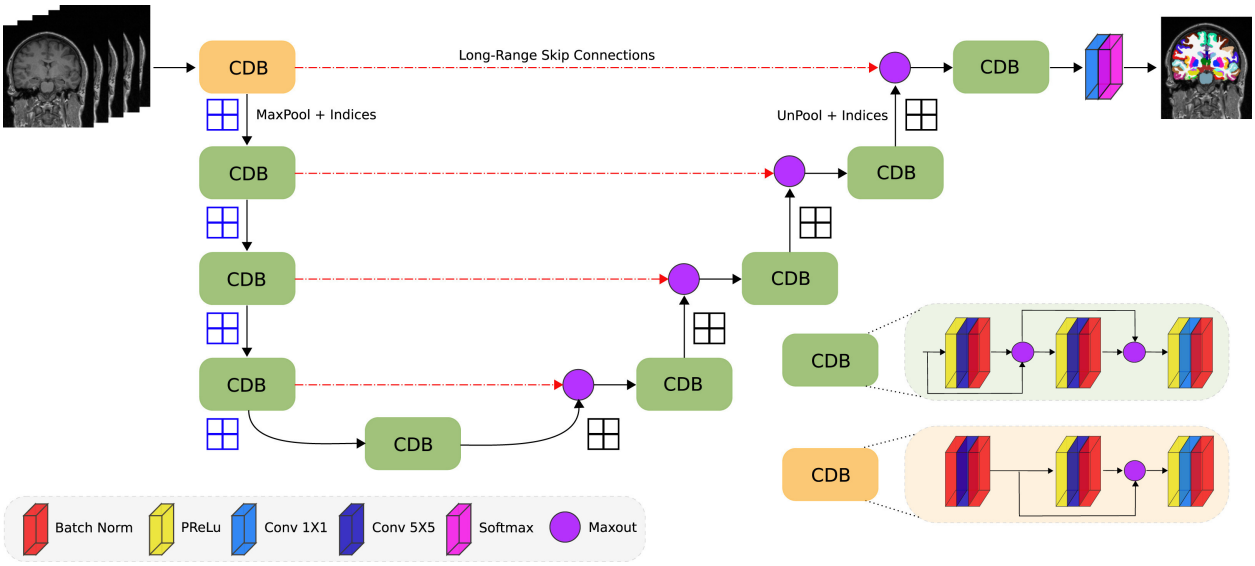


Figure 8: Illustration of FastSurfer's architecture. The CNN consists of four competitive dense blocks (CDB) in the encoder and decoder part, separated by a bottleneck layer. Figure reproduced from (Henschel et al., 2020).

A.3 Statistical Analysis of Harmonic Series Variance Homogeneity

Descriptive Statistics by Method				
Method	**Mean Std. Dev.**	**95% CI**	**F-statistic vs. PRISM SR**	**p-value**
PRISM SR	1.65×10^{-4}	$[1.08, 2.23] \times 10^{-4}$	—	—
MCA RR	2.41×10^{-4}	$[1.37, 3.45] \times 10^{-4}$	$F = 1.58$	$p = 0.210$
Verrou SR	1.48×10^{-4}	$[1.00, 1.96] \times 10^{-4}$	$F = 0.21$	$p = 0.651$
FM SR	2.10×10^{-4}	$[1.27, 2.94] \times 10^{-4}$	$F = 0.78$	$p = 0.380$

Table 2: Statistical analysis of variance homogeneity across stochastic rounding methods in harmonic series computation. Levene's test confirms homogeneity of variances across stochastic rounding implementations ($F = 1.26$, $p = 0.29$), supporting the validity of comparative analyses. All pairwise F-tests compare against PRISM SR as the reference method. Tests performed at $\alpha = 0.05$ significance level.