An Investigation into the Sources, Impact, and Mitigation of Computational Nondeterminism in Deep Learning

1 The Imperative for Determinism in Computational Science and Deep Learning

1.1 The Reproducibility Crisis as a Scientific Context

Over the past decade, a significant discourse has emerged across scientific disciplines regarding a "reproducibility crisis"[1]. Surveys indicate a widespread belief among scientists that science is facing such a crisis, with a substantial portion of published research considered difficult or impossible to reproduce[2]. This challenge to the reliability of the scientific enterprise is not primarily rooted in deliberate misconduct like data fabrication, which accounts for a very small fraction of cases[2]. Instead, the crisis stems from a collection of more subtle, heterogeneous factors, including biases in hypothesis testing, poor experimental design, statistical problems, and a pervasive lack of transparency in the reporting of methods, data, and analysis[1].

The response to this crisis has been a call for greater methodological rigor, with frameworks such as the "five pillars of reproducible computational research"—literate programming, version control, environment control, persistent data sharing, and documentation—being proposed to ensure that scientific work can be verified and built upon[4]. This context underscores the fundamental importance of reproducibility for maintaining the trustworthiness and cumulative nature of scientific progress[5].

1.2 Defining the Scope: Computational Reproducibility vs. Experimental Replicability

Within this broader crisis, it is critical to distinguish between two related but distinct concepts: **replicability** and **reproducibility**[1]. Replicability generally refers to the ability to redo an entire experiment, often with new data or under different conditions, and obtain results that are consistent with the original study. This is a cornerstone of the scientific method, validating the robustness of a finding.

Reproducibility, particularly in computational disciplines, refers to a more fundamental and seemingly simpler standard: the ability to obtain the exact same results from the same dataset using the same source code and computational environment[1]. This is often termed "computational reproducibility" and represents a baseline for verification[1].

While many scientific fields grapple with the complexities of experimental replicability, the domain of deep learning often fails at the more basic prerequisite of computational reproducibility. The inability to guarantee that the same code, acting on the same data, will produce a bit-for-bit identical output on successive runs presents a profound challenge. This failure undermines essential scientific and engineering practices, making it exceedingly difficult to debug models, audit results for safety-critical applications, or reliably attribute changes in performance to specific modifications in an experiment[6].

1.3 The Acute Problem in ML-Based Science

The challenge of reproducibility is particularly acute in machine learning (ML)-based science[7]. The inherent complexity of ML code, a historical lack of standardization in software development practices, and the prevalence of subtle pitfalls like data leakage have created a research environment where reproducibility failures are common[7]. This situation is exacerbated by the fact that the computational tools themselves often behave in a non-deterministic manner.

Therefore, the problem of nondeterminism in deep learning is not merely an inconvenience but a fundamental barrier to scientific rigor. It introduces an uncontrolled variable into every experiment, confounding results and eroding confidence in the findings. For developers, nondeterministic outputs make debugging a Sisyphean task, as a bug may not manifest consistently[6]. In high-stakes domains such as autonomous vehicles or medical diagnostics, the inability to perform a bit-exact reproduction of a specific inference result for auditing or failure analysis is unacceptable[8].

The work proposed herein directly confronts this foundational challenge, seeking to understand and control the sources of nondeterminism that threaten the scientific validity of deep learning research. The problem is not one of poor scientific practice, but of the fundamental, and often unpredictable, behavior of the computational instruments upon which the field relies.

2 Fundamental Sources of Computational Nondeterminism

The nondeterminism observed in deep learning systems is not the result of a single flaw but is an emergent property arising from the interaction between the fundamental nature of computer arithmetic and the architectural principles of modern parallel hardware. While both the rules of floating-point arithmetic and the operation of a GPU are deterministic in isolation, their combination at the scale required for deep learning creates systemic unpredictability.

2.1 The Illusion of Exactness: Floating-Point Arithmetic and Non-Associativity

Modern scientific computing is built upon the IEEE 754 standard for floating-point arithmetic, which defines standardized formats for representing real numbers[9]. These formats, such as the common 32-bit single-precision (float) and 64-bit double-precision (double), encode a number using a sign bit, an exponent, and a significand (or mantissa)[10]. The core limitation of this system is that a finite number of bits must be used to approximate an infinite set of real numbers.

Consequently, many values cannot be represented exactly. For instance, a simple fraction like $2/3$ has an infinite repeating binary representation and must be rounded to fit into the finite precision of the significand[10]. This rounding error is described as the "characteristic feature of floating-point computation"[12].

A critical and often counter-intuitive consequence of this finite precision is that floating-point addition is **non-associative**. In pure mathematics, the associative law states that:

$$(a + b) + c = a + (b + c)$$

However, in floating-point arithmetic, this property does not hold[6]. A classic example of this is "catastrophic cancellation". Let $a = 10^{20}$, $b = -10^{20}$, and $c = 3.14$.

- The operation $(a + b) + c$ evaluates to $(10^{20} - 10^{20}) + 3.14 = 0 + 3.14 = 3.14$.
- The operation $a + (b + c)$ evaluates to $10^{20} + (-10^{20} + 3.14) = 0$.

Due to the limited precision of the floating-point representation, the small value of 3.14 is lost when added to -10^{20}, resulting in the intermediate sum being rounded to -10^{20}. The final operation becomes $10^{20} - 10^{20} = 0$. This demonstrates that the order of operations can

drastically alter the final result. This behavior is not a flaw but a well-understood and deliberate trade-off made to enable the high-performance hardware implementation of floating-point units[9]. For any fixed sequence of operations, the IEEE 754 standard guarantees a deterministic, bit-identical result; the problem arises when the sequence itself is not fixed[12].

2.2 The Unpredictability of Parallelism: GPU Architecture and Reduction Operations

Modern deep learning is enabled by the massive parallelism of Graphics Processing Units (GPUs). GPUs achieve their performance by executing thousands of lightweight threads concurrently, organized into a hierarchy of thread blocks[6]. A fundamental computational pattern in deep learning is the **reduction**, where an array of values is reduced to a single scalar via an operation like summation, averaging, or finding the maximum[15]. On a GPU, this is typically implemented using a tree-based approach: threads within a block compute partial sums of subsets of the data, and these partial sums are then further combined until a final result is produced[16].

Nondeterminism is introduced because the execution order of these parallel operations is not guaranteed. The order in which threads or thread blocks complete their tasks and their partial results are accumulated can vary from one run to the next, depending on the dynamic state of the GPU scheduler and resource availability[6]. This variable execution order, when combined with the order-dependent nature of non-associative floating-point addition, results in numerically different, non-deterministic final results for the same reduction operation.

To manage race conditions where multiple threads write to the same memory location, programmers can use **atomic operations** (e.g., `atomicAdd`), which ensure that read-modify-write cycles are completed without interruption[6]. While this guarantees a correct (though not necessarily bit-identical) result, it forces parallel threads into a serialized queue, creating a severe performance bottleneck that negates the primary advantage of the parallel architecture[19]. Furthermore, a key design choice in modern GPUs is the absence of a low-cost, global synchronization mechanism across all thread blocks on the device. This is an intentional decision to avoid the high hardware cost and potential for deadlock that such a feature would entail[16]. This architectural constraint means that large-scale reductions must often be decomposed into multiple, separate kernel launches, with the launch itself acting as a slow, implicit point of global synchronization, further complicating any attempt to enforce a fixed execution order.

3 Manifestations of Nondeterminism in Deep Learning Systems

The fundamental principles of floating-point non-associativity and parallel execution scheduling manifest as concrete sources of nondeterminism at multiple levels of the deep learning stack. These range from low-level computational kernels to high-level framework behaviors, all contributing to the challenge of achieving reproducible results.

3.1 Kernel-Level Nondeterminism and the Batch Invariance Problem

The core mathematical operations in deep learning models, including matrix multiplication (matmul), convolution, normalization, and attention, are all intensive users of parallel reduction operations[6]. A key insight is that even when a model is configured for deterministic behavior (e.g., using greedy decoding with a temperature of 0), the output for a single, identical input can vary between runs. This phenomenon arises from a lack of **batch invariance**: the computational result for one input is dependent on the other inputs it is processed with in the same batch[6].

This problem can be understood through a clear causal chain:

1. **Dynamic Batching:** To maximize hardware utilization and throughput, inference servers dynamically group incoming user requests into batches. The size of these batches is therefore variable and dependent on the server's load at any given moment[6].
2. **Adaptive Kernel Strategies:** Highly optimized libraries like NVIDIA's cuBLAS employ different internal parallelization strategies for operations like matrix multiplication depending on the shapes of the input matrices. For example, a "Split-K" strategy may be used to increase parallelism for certain matrix dimensions, which are directly influenced by the batch size[6].
3. **Variable Reduction Orders:** These different parallelization strategies result in different tiling and chunking of the computation, which in turn changes the order in which partial products are summed (reduced) to produce the final output elements[6].
4. **Nondeterministic Outputs:** This change in reduction order directly interacts with floating-point non-associativity.

Consequently, the final numerical result of a matmul or convolution operation for a specific input vector can have bitwise differences depending on the size of the batch it was processed in[6]. This affects nearly all major components of a modern neural network, including RMSNorm (which reduces over the feature dimension), matrix multiplication, and attention mechanisms (which reduce over both feature and sequence dimensions)[6].

3.2 Framework-Level Nondeterminism: The Software Stack's Contribution

On top of the hardware- and kernel-level issues, deep learning frameworks like PyTorch and TensorFlow introduce additional sources of nondeterminism. These are often default behaviors designed to maximize performance.

- **cuDNN Algorithm Benchmarking:** The NVIDIA CUDA Deep Neural Network (cuDNN) library provides highly optimized routines for convolutions. By default, frameworks often enable `torch.backends.cudnn.benchmark = True`. When a convolution is performed on a new input shape for the first time, cuDNN will benchmark several different algorithms and select the fastest one for all subsequent calls with that shape. This selection process can be nondeterministic, and the chosen algorithm may itself be nondeterministic, leading to run-to-run variation[20].
- **Parallel Data Loading:** The `DataLoader` classes in frameworks commonly use multiple worker processes (`num_workers > 0`) to load and preprocess data in parallel to prevent data loading from becoming a bottleneck. However, the order in which these parallel workers complete their tasks and feed data to the GPU is not guaranteed, which can introduce nondeterminism in the sequence of training batches[20].
- **Other Nondeterministic Operations:** Certain operations are known to have nondeterministic implementations on CUDA for performance reasons. Examples include `torch.bmm` when used with sparse tensors and certain implementations of RNN and LSTM layers in specific CUDA versions[21].

Frameworks offer global flags, such as `torch.use_deterministic_algorithms(True)`, to try and control these behaviors. While useful, these flags often act as a blunt instrument, forcing the framework to fall back to slower, generic, and guaranteed-deterministic algorithms. This can result in severe performance degradation, with some users reporting slowdowns of up to 100x, making these flags impractical for large-scale training or performance-critical inference[8]. This highlights the need for more granular and efficient solutions that can provide determinism without sacrificing performance entirely.

4 Experimental Investigation and Results

To move beyond a general acknowledgment of nondeterminism, I conducted a systematic experimental investigation guided by three Research Questions (RQs). These experiments isolate the sources of nondeterminism in modern Transformer architectures and evaluate the engineering trade-offs of potential mitigation strategies.

The complete codebase for reproducing these experiments, including the data generation scripts for RQ1-RQ3 and the analysis notebooks, is hosted in project repository[26]. All experiments were conducted on a single NVIDIA GeForce RTX 3070 (8 GB GDDR6) using PyTorch $\geq$ 2.1, Transformers $\geq$ 4.39, and Triton $\geq$ 2.2.

4.1 RQ1: Magnitude and Origin of Inference Nondeterminism

Research Question: How does parallel batch processing introduce computational nondeterminism in Transformer-based LLMs, and is it possible to achieve bitwise identity between batched and solo inference?

4.1.1 Methodology

Implementation of Batch Invariant Kernels: To enforce deterministic execution, I utilized and adapted the `batch_invariant_ops` library developed by Thinking Machines Lab[25]. While the original library provides a broad set of deterministic overrides, I modified the codebase to better suit modern PyTorch environments and my specific experimental needs:

- **Hardware Abstraction:** I updated the hardware detection logic to utilize the modern `torch.accelerator` API, ensuring compatibility with newer PyTorch versions and diverse hardware backends (e.g., XPU) beyond standard CUDA devices.
- **Targeted Overrides:** I refined the scope of operator overrides. The original library enforces determinism on a wide range of operations including batch matrix multiplication (`bmm`) and activations (`silu`). I found that strict serialization of `bmm` introduced excessive overhead without being the primary source of divergence in my specific testbed. Consequently, my implementation strictly targets the most critical sources of floating-point non-associativity: persistent matrix multiplication (`mm`, `addmm`) and reduction operations (`log_softmax`, `mean`).
- **Kernel Configuration:** I introduced specific configuration hooks (`AttentionBlockSize`) to tune Triton kernel parameters for the specific tensor shapes encountered in GPT-2 inference.

Experimental Protocol: I hypothesized that the primary driver of nondeterminism in Transformers is the interaction between batch-dependent memory layouts and the non-associative reductions in the attention mechanism. To test this, I utilized a GPT-2 Large (774M parameters) model loaded in bfloat16 precision. I selected 10 diverse natural language prompts (e.g., "The quick brown fox...", "Quantum mechanics describes...") and processed them in two contexts:

1. **Solo Inference:** The prompt is processed alone (batch size = 1).
2. **Batched Inference:** The same prompt is processed alongside a "distractor" sequence (repeated 5x to force padding) to create a batch size of 2.

I extracted the logit vector at the last real token position and computed the maximum element-wise absolute difference: $\text{Diff} = \max_j |L_{\text{solo}}[j] - L_{\text{batched}}[j]|$. This procedure was repeated using both standard PyTorch eager execution (which uses cuBLAS/cuDNN) and a custom "Batch Invariant" implementation using Triton kernels designed to enforce fixed-order accumulation.

4.1.2 Results

The results, summarized in Table 2, reveal a significant divergence in standard execution.

The average divergence of **0.175** in bfloat16 is substantial. Since bfloat16 possesses only ∼3 significant decimal digits, a deviation of this magnitude can alter the argmax of the softmax distribution, leading to different token generation. However, the custom Batch Invariant kernels achieved exactly **0.0** difference, confirming that nondeterminism is not inherent to the hardware but is a consequence of the reduction order chosen by standard kernels.

4.2 RQ2: The Performance Cost of Deterministic Execution

Research Question: What is the latency overhead associated with enforcing strict deterministic execution compared to standard optimized kernels?

4.2.1 Methodology

Having established that bitwise determinism is possible (RQ1), I sought to quantify its cost. I benchmarked the inference latency (ms per forward pass) of GPT-2 Large under three configurations:

- **Eager Mode:** Standard PyTorch execution (non-deterministic).
- **SDPA Mode:** Scaled Dot-Product Attention (Flash Attention), a highly optimized fused kernel (non-deterministic).
- **Batch Invariant Mode:** The custom deterministic kernels from RQ1.

I performed 20 iterations with a batch size of 8. To contrast the overhead observed in Transformers, I benchmarked a standard Convolutional Neural Network (CNN) layer. I measured the forward pass latency of a single `Conv2d` layer (64 channels, 3×3 kernel, 32×32 input) in "Fast" mode (cuDNN benchmarking enabled) versus "Deterministic" mode (`torch.use_deterministic_algorithms(True)`).

4.2.2 Results

The performance penalty for strict determinism in Transformers was found to be prohibitive. Table 3 details the latency measurements for 10 separate runs, comparing standard eager execution, Flash Attention (SDPA), and the custom deterministic kernel.

The deterministic implementation was approximately **26x slower** than standard execution. This extreme overhead arises because deterministic kernels must serialize reductions (negating parallelism) and cannot utilize fused operations like Flash Attention, which rely on non-deterministic block-wise reductions.

Table 4 demonstrates that the cost of determinism in CNNs is far more moderate (or even non-existent, as seen in batch size 16 where the deterministic algorithm happened to be faster) compared to the ∼2500% overhead in Transformers. This is because convolution operations involve fewer reduction dimensions than the $O(N^2)$ attention mechanism, and cuDNN provides dedicated deterministic algorithms for common convolution shapes that are highly optimized.

4.3 RQ3: Efficacy of Stochastic Rounding for Training Stability

Research Question: To what extent can stochastic rounding mitigate output variance and improve training stability in low-precision settings?

4.3.1 Methodology

Given the prohibitive cost of strict determinism for Transformers (RQ2), I investigated **stochastic rounding** as a numerical mitigation strategy. I trained a ResNet-18 model on CIFAR-10 from scratch using simulated 16-bit fixed-point arithmetic under three conditions:

1. **FP32:** Full 32-bit floating-point precision (Baseline).
2. **Nearest:** 16-bit quantization with standard round-to-nearest.
3. **Stochastic:** 16-bit quantization with stochastic rounding.

4.3.2 Results

Stochastic rounding outperformed both the low-precision and full-precision baselines in this context. Table 5 summarizes the final training metrics.

To visualize the training dynamics, Table 6 presents the validation accuracy progression over 10 epochs.

Analysis:

- **Accuracy:** Stochastic rounding achieved the highest final accuracy (72.06%), surpassing the FP32 baseline by 2.35 percentage points. This suggests that the noise injected by stochastic rounding acts as a form of implicit regularization, preventing the overfitting observed in the FP32 run (where accuracy dropped from epoch 9 to 10).
- **Stability:** While the "Nearest" method suffered from gradient stagnation in early epochs (lagging ∼5-7% behind FP32), Stochastic rounding maintained a trajectory closer to FP32, eventually overtaking it.
- **Cost:** The stochastic method introduced a ∼14% training time overhead (232s vs 203s) due to the computational cost of random number generation. This is a negligible cost compared to the 2500% overhead of strict determinism seen in RQ2.

5 Discussion

The results of this investigation highlight a fundamental hardware-software tension in deep learning: **efficiency is currently inextricable from nondeterminism**.

- **The Cost of Order:** The "Illusion of Exactness" discussed in Section 2.1 is shattered by the scale of deep learning. While I *can* force the hardware to behave deterministically (RQ1), doing so requires fighting against the GPU's massive parallelism (RQ2). A 2500% slowdown is unacceptable for production systems, meaning that for the foreseeable future, inference at scale will remain non-deterministic.
- **Statistical vs. Bitwise Stability:** The success of stochastic rounding (RQ3) suggests a paradigm shift. Rather than chasing the phantom of bitwise identity—which is expensive and fragile—researchers should prioritize *statistical consistency*. Stochastic rounding ensures that while individual operations may vary, the aggregate training dynamics remain unbiased and stable.
- **Implications for Reproducibility:** The "reproducibility crisis" in AI cannot be solved solely by code versioning. If the hardware itself introduces variance, my definition of reproducibility must evolve from "identical bits" to "statistically indistinguishable distributions."

6 Conclusions

The problem of computational nondeterminism in deep learning is a significant and multifaceted challenge that strikes at the heart of scientific reproducibility. It is not a superficial issue of coding practice but an emergent property arising from the fundamental interaction between the non-associativity of floating-point arithmetic and the unpredictable execution order of massively parallel hardware. This report has outlined the primary sources of this nondeterminism, from kernel-level batch invariance problems to framework-level behaviors, providing a comprehensive taxonomy of the issue.

This research set out to quantify and mitigate computational nondeterminism in deep learning. I have demonstrated that the issue is not merely theoretical but manifests as measurable, context-dependent divergence in model outputs (RQ1). I have established that the engineering cost of eliminating this divergence through strict serialization is currently prohibitive for Transformer architectures (RQ2). However, I have also identified a viable path forward: stochastic rounding offers a practical, high-performance alternative that ensures numerical stability without the crushing overhead of bitwise determinism (RQ3).

Future work should focus on optimizing deterministic kernels to reduce the 25x gap and exploring how stochastic rounding interacts with other regularization techniques in large-scale model training. For now, practitioners must accept that in the realm of deep learning, the same input may not always yield the same output, and design their systems to be robust to this inherent uncertainty.

References

[1] Reproducibility of Scientific Results (Stanford Encyclopedia of ...). Accessed October 25, 2025. https://plato.stanford.edu/entries/scientific-reproducibility/

[2] Is science really facing a reproducibility crisis, and do we need it to ... Accessed October 25, 2025. https://www.pnas.org/doi/10.1073/pnas.1708272114

[3] "Reproducible" research in mathematical sciences requires changes in our peer review culture and modernization of our current publication approach - NIH. Accessed October 25, 2025. https://pmc.ncbi.nlm.nih.gov/articles/PMC6240027/

[4] Five pillars of computational reproducibility: bioinformatics and ... Accessed October 25, 2025. https://academic.oup.com/bib/article/24/6/bbad375/7326135

[5] Improving reproducibility in computational biology research - Research journals - PLOS. Accessed October 25, 2025. https://journals.plos.org/ploscompbiol/article?id=10.1371/journal.pcbi.1007881

[6] Defeating Nondeterminism in LLM Inference.pdf (Source File)

[7] Leakage and the Reproducibility Crisis in ML-based Science. Accessed October 25, 2025. https://reproducible.cs.princeton.edu/

[8] Determinism in Deep Learning (S9911) — NVIDIA. Accessed October 25, 2025. https://developer.download.nvidia.com/video/gputechconf/gtc/2019/presentation/s9911-determinism-in-deep-learning.pdf

[9] Effects of Floating-Point non-Associativity on Numerical Computations on Massively Multithreaded Systems. Accessed October 25, 2025. https://www.sci.utah.edu/~beiwang/teaching/cs6210-fall-2016/nonassociativity.pdf

[10] 1. Introduction — Floating Point and IEEE 754 13.0 documentation. Accessed October 25, 2025. https://docs.nvidia.com/cuda/floating-point/index.html

[11] Floating-point arithmetic - Wikipedia. Accessed October 25, 2025. https://en.wikipedia.org/wiki/Floating-point_arithmetic

[12] What Every Computer Scientist Should Know About Floating-Point ... Accessed October 25, 2025. https://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html

[13] Is floating-point addition and multiplication associative? - Stack Overflow. Accessed October 25, 2025. https://stackoverflow.com/questions/10371857/is-floating-point-addition-and-multiplication-associative

[14] Example of non associative floating point addition - Stack Overflow. Accessed October 25, 2025. https://stackoverflow.com/questions/46769671/example-of-non-associative-floating-point-addition

[15] 7 Step Optimization of Parallel Reduction with CUDA — by Rimika Dhara — Medium. Accessed October 25, 2025. https://medium.com/@rimikadhara/7-step-optimization-of-parallel-reduction-with-cuda-33a3b2feafd8

[16] Optimizing Parallel Reduction in CUDA — NVIDIA Developer ... Accessed October 25, 2025. https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf

[17] IMPLEMENTATIONS OF THE PARALLEL SUM IN CUDA. NON-DETERMINISTIC... - ResearchGate. Accessed October 25, 2025. https://www.researchgate.net/figure/IMPLEMENTATIONS-OF-THE-PARALLEL-SUM-IN-CUDA-NON-DETERMINISTIC-IMPLEMENTATIONS-SHOWN-IN_tbl2_387848485

[18] GPU MODE Lecture 9: Reductions - Christian Mills. Accessed October 25, 2025. https://christianjmills.com/posts/cuda-mode-notes/lecture-009/

[19] Optimizing the GPU kernel — CUDA training materials documentation. Accessed October 25, 2025. https://enccs.github.io/cuda/3.01_ParallelReduction/

[20] Reproducible Deep Learning Using PyTorch — by Darina Bal Roitshtain — Medium. Accessed October 25, 2025. https://darinabal.medium.com/deep-learning-reproducible-results-using-pytorch-42034da5ad7

[21] Reproducibility — PyTorch 2.9 documentation. Accessed October 25, 2025. https://docs.pytorch.org/docs/stable/notes/randomness.html

[22] How to get deterministic behavior? - PyTorch Forums. Accessed October 25, 2025. https://discuss.pytorch.org/t/how-to-get-deterministic-behavior/18177

[23] How do you compare different deep learning experiments when determinism is so hard to achieve? - Reddit. Accessed October 25, 2025. https://www.reddit.com/r/deeplearning/comments/1avfz63/how_do_you_compare_different_deep_learning/

[24] Deep Learning with Limited Numerical Precision - arXiv. Accessed October 25, 2025. https://arxiv.org/abs/1502.02551

[25] Thinking Machines Lab. batch_invariant_ops. GitHub Repository. Accessed December 7, 2025. https://github.com/thinking-machines-lab/batch_invariant_ops

[26] XYCrus. QuantifyNondeterminism. GitHub Repository. Accessed December 10, 2025. https://github.com/XYCrus/QuantifyNondeterminism

Table 1: A Taxonomy of Nondeterminism Sources in Deep Learning

Level	**Source**	**Mechanism**	**Affected DL Operations**
Numerical	Floating-Point Non-Associativity	Limited precision of IEEE 754 standard causes rounding errors, making the order of additions significant: $(a + b) + c \neq a + (b + c)$.	Summation, Dot Product, Norm, Softmax (any reduction)
Hardware	Parallel Reduction Scheduling	Unpredictable completion order of parallel threads/blocks on a GPU leads to a variable order of accumulation for partial results.	Matrix Multiplication, Convolution, Normalization Layers, Attention
System	Dynamic Batching (Batch Invariance)	Variable server load leads to different batch sizes, causing kernels to select different parallelization strategies with different reduction orders.	All batched operations in a deep learning model forward pass.
Framework	cuDNN Algorithm Selection	Benchmarking feature (`cudnn.benchmark=True`) non-deterministically selects the fastest convolution algorithm, which may itself be nondeterministic.	Convolution (`nn.Conv2d`)
Framework	Parallel Data Loading	Multiple worker processes (`num_workers > 0`) load data in a non-guaranteed order, affecting the sequence of training batches.	Data loading and augmentation pipelines (`DataLoader`)

Table 2: RQ1 Results: Logit Divergence (Max Absolute Difference)

Run	Prompt Excerpt	Diff (Standard)	Diff (Invariant)	Success
1	"The quick brown fox..."	0.250	0.0	✓
2	"Artificial intelligence..."	0.125	0.0	✓
3	"To be or not to be..."	0.125	0.0	✓
4	"The grand unifying..."	0.125	0.0	✓
5	"Python is a popular..."	0.125	0.0	✓
6	"In the beginning God..."	0.250	0.0	✓
7	"A journey of a thousand..."	0.125	0.0	✓
8	"Quantum mechanics..."	0.250	0.0	✓
9	"The stock market..."	0.125	0.0	✓
10	"Climate change poses..."	0.250	0.0	✓
Avg	—	**0.175**	**0.0**	**100%**

Table 3: RQ2 Results: Latency Comparison (ms per forward pass)

Run	Eager (ms)	Invariant (ms)	SDPA (ms)	vs Eager	vs SDPA
1	35.4	1214.1	27.1	+3332%	+4375%
2	34.2	860.5	27.5	+2416%	+3025%
3	34.1	841.7	24.4	+2368%	+3350%
4	34.0	847.4	24.2	+2390%	+3401%
5	34.1	929.1	24.2	+2623%	+3736%
6	34.2	845.4	26.6	+2370%	+3076%
7	34.5	845.1	27.5	+2348%	+2978%
8	34.6	1016.3	24.7	+2834%	+4017%
9	34.3	845.3	27.7	+2363%	+2957%
10	34.5	845.6	24.5	+2353%	+3346%
Avg	**34.4**	**909.1**	**25.8**	**+2540%**	**+3426%**

Table 4: RQ2 Complementary Results: CNN Convolution Latency

Batch Size	Fast (ms)	Deterministic (ms)	Overhead
1	0.055	0.077	+39.4%
16	0.141	0.124	-12.1%
64	0.442	0.549	+24.4%

Table 5: RQ3 Results: Training Performance Comparison

Mode	Best Val Acc (%)	Final Val Acc (%)	Training Time (s)
FP32 (Baseline)	71.03	69.71	203.1
Nearest (16-bit)	71.26	71.26	209.5
Stochastic (16-bit)	**72.06**	**72.06**	232.1

Table 6: Validation Accuracy Trajectory per Epoch

Epoch	**FP32**	**Nearest**	**Stochastic**
1	41.17%	36.57%	37.50%
2	53.36%	45.93%	50.59%
3	58.84%	50.61%	59.13%
4	65.19%	57.01%	64.31%
5	66.62%	59.85%	64.43%
6	65.25%	65.72%	66.73%
7	69.84%	68.63%	64.78%
8	68.87%	68.50%	70.55%
9	71.03%	66.63%	69.47%
10	69.71%	71.26%	**72.06%**