

Identical Outputs, Different Gradients: A Floating-Point Counterexample in PyTorch and JAX

Anonymous

22 October 2026; version 1.0.0

Abstract

Two programs can return the same floating-point values and still send different signals to an optimizer. We explain a small binary32 counterexample from *Homotopy Fibers and Observation-Preserving Semantics for Finite-Precision Training* [1] and reproduce it with the built-in reverse-mode rules of PyTorch and JAX on CPU. Both expressions return zero throughout a specified interval. Their automatic gradients are one and zero, respectively, so one SGD update separates their parameters. The numerical mechanism is ordinary rounding near a large power of two. Our contribution is an accessible explanation, direct framework reproduction, and a separately recorded compilation check. The resulting contract is practical: a claim that preserves training behavior must specify more than forward outputs.

1. The example in one minute

Run this complete example with the CPU PyTorch environment described in the source bundle:

```
import torch

C = torch.tensor(2**30, dtype=torch.float32)
x = torch.tensor(0.0, dtype=torch.float32, requires_grad=True)
h = (C + x) - C
z = x - x
dh, = torch.autograd.grad(h, x)
dz, = torch.autograd.grad(z, x)
print(h.item(), z.item(), dh.item(), dz.item())
```

The recorded result is 0.0 0.0 1.0 0.0. No custom gradient, gradient hook, detached tensor, or external derivative is used. Both expressions depend syntactically on `x`; this makes a zero reverse result directly observable through the same autograd interface.

Program	Output at zero	Automatic gradient	Parameter after one SGD update
<code>h = (C + x) - C</code>	0	1	-1
<code>z = x - x</code>	0	0	0

Both parameters start at zero and the learning rate is one. The rate is chosen to make the arithmetic transparent. This is a scalar demonstration rather than a benchmark of a learning task.

Figure 1 in the attached PDF and offline reading edition puts the rounded outputs, automatic gradients, and eight-update parameter traces side by side. The source bundle includes the standalone SVG and the data used to draw it.

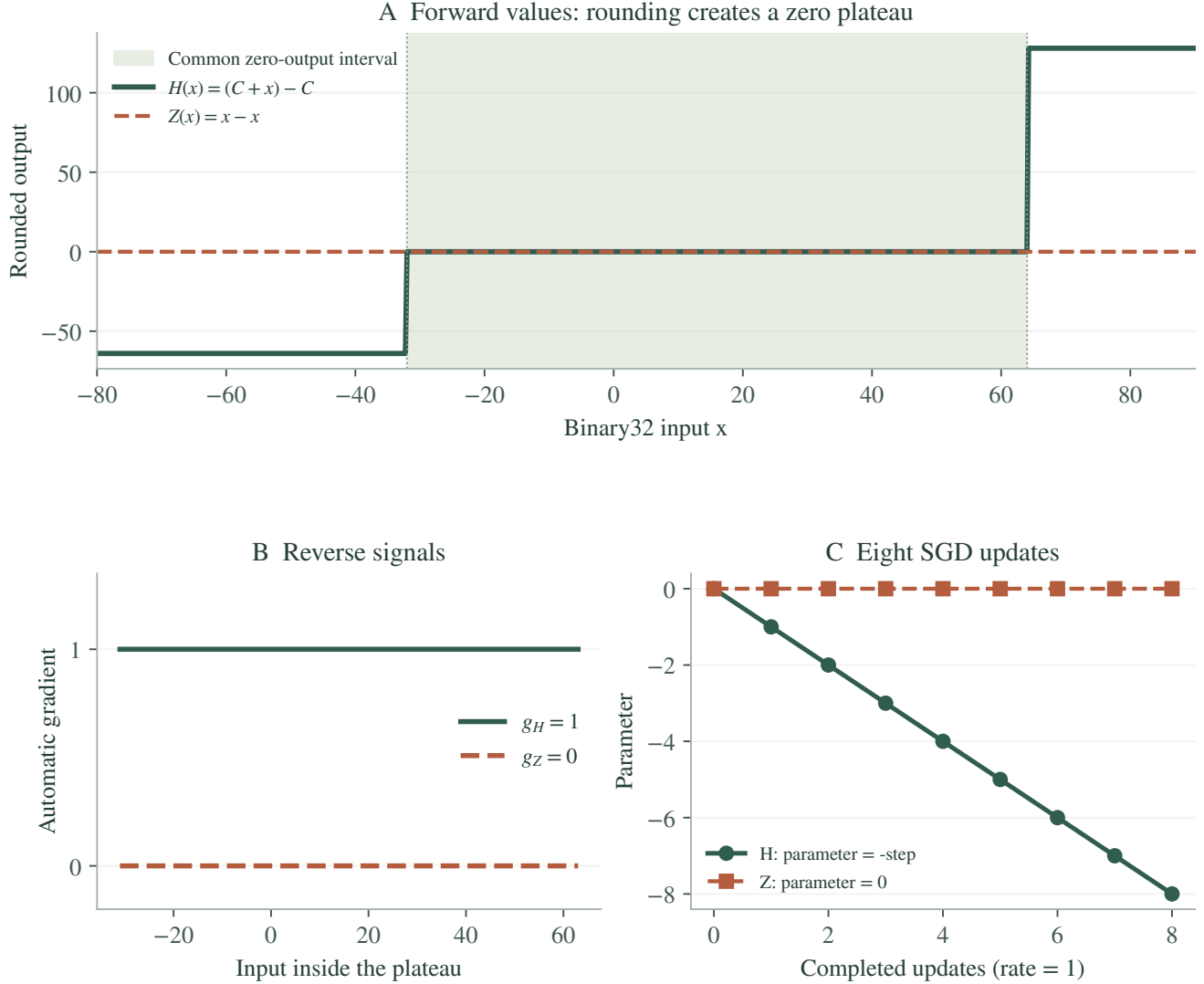


Figure 1: Equal rounded outputs, different reverse signals, and different parameter traces. Panel A uses separately rounded NumPy binary32 operations; panels B and C show the matching PyTorch-eager and JAX-op-by-op results. The shaded region is the open interval in Proposition 2.1. Whole-expression JAX JIT is recorded separately in Section 6.

2. Where the unit change goes

Let fl mean a separately rounded binary32 operation with round-to-nearest, ties-to-even. Fix

$$C = 2^{30}, \quad H(u) = \text{fl}(\text{fl}(C + u) - C), \quad Z(u) = \text{fl}(u - u).$$

The number C is exactly representable. Binary32 has 24 significant binary digits, including the implicit leading digit. The representable number immediately below C is $C - 64$; the number immediately above is $C + 128$. The spacing differs on the two sides because C is the start of a new binade.

Proposition 2.1 (Equal rounded outputs). For every binary32 input u with $-32 < u < 64$, separate evaluation of H and Z gives zero.

Proof. The midpoints between C and its two neighbors are $C - 32$ and $C + 64$. Thus $C + u$ rounds to C throughout the stated open interval. The second operation subtracts C from itself. In Z , subtraction of the same finite input from itself is exact. Both results are zero. $\square$

The bound is asymmetric. In the recorded configuration, $H(-33) = -64$ and $H(65) = 128$. At the two midpoint ties, $H(-32) = H(64) = 0$ because ties-to-even selects C . We use the open interval in the proposition to state the mechanism without relying on those endpoint cases.

This elementary rounding argument is already present in [1]. The same power-of-two example supplies the scientific starting point here; we do not claim a new floating-point pathology.

3. Why the automatic gradients differ

Throughout this article, *automatic gradient* means the reverse-mode value returned by the framework for the executed expression graph.

PyTorch records the operations that produced a tensor and applies their reverse rules [2]. In the ordinary rules for addition and subtraction, a constant offset passes the incoming cotangent to its variable input. Thus the two operations in H pass a seed of one back to u . In Z , the two occurrences of u receive contributions of one and minus one, which sum to zero. The framework runs below measure these rules directly.

Proposition 3.1 (Different reverse signals). Under these primitive reverse rules, $g_H(u) = 1$ and $g_Z(u) = 0$ on the domain of Proposition 2.1.

Proof. Reverse composition in H sends 1 through subtraction of a constant, then addition of a constant, leaving 1. Reverse accumulation in Z gives $1 - 1 = 0$. $\square$

Over exact real arithmetic, the expression $(C + u) - C$ equals u , whose derivative is one. Its separately rounded forward evaluation instead has a zero plateau. A real, piecewise-constant extension of that rounded evaluation has derivative zero inside the plateau. These are different specifications of differentiation. The measured reverse signal follows the recorded primitive rules; recovering the classical derivative of the rounded step function is a separate task.

Corollary 3.2 (Forward values do not determine the reverse signal). No function of the restricted rounded forward map alone recovers both programs’ prescribed reverse signal at zero.

Proof. Proposition 2.1 gives the same restricted forward map for H and Z . Such a recovery function would assign a single answer to that map. Proposition 3.1 requires both one and zero. $\square$

This is the obstruction developed more generally in [1]. The distinction becomes visible before introducing the higher-categorical construction.

4. What the framework runs establish

The measured environment is Python 3.12.3, NumPy 2.5.4, PyTorch 2.14.1+cpu, JAX 0.12.0.1, and jaxlib 0.12.0.1 on Linux x86_64 CPU. The main witness uses PyTorch eager execution and JAX op-by-op execution, explicit binary32 tensors, and built-in reverse rules. All 52 assertions in the full framework verification run passed.

The verification script checks all 95 integer inputs from -31 through 63 in each framework. On every sampled input, the two forward outputs are bitwise equal binary32 zeros, while the automatic gradients are one and zero. Nine additional input checks record both rounding boundaries and values just outside them. The written proof establishes the interval claim; sampling checks the implementations on the recorded cases.

Input	H in PyTorch eager	H in JAX op-by-op	Z in both
-33	-64	-64	0
-32	0	0	0
0	0	0	0
64	0	0	0
65	128	128	0

The JAX reproduction uses the same two expressions:

```
import jax
import jax.numpy as jnp

C = jnp.array(2**30, dtype=jnp.float32)
x = jnp.array(0.0, dtype=jnp.float32)
h, dh = jax.value_and_grad(lambda u: (C + u) - C)(x)
z, dz = jax.value_and_grad(lambda u: u - u)(x)
print(float(h), float(z), float(dh), float(dz))
```

Run on the CPU without wrapping the whole expression in `jax.jit`. This is *op-by-op* execution: JAX still uses compiled primitives internally. The distinction is whether the entire expression is available to one compilation and optimization pass.

Two controls help interpret the result. First, with binary64 inputs and constants, $H(1) = 1$ in both frameworks; the unit change is resolved at this magnitude. Second, a central finite difference of the binary32 forward program at zero with step one returns zero, because both sampled outputs lie on its plateau. The automatic gradient is still one. Precision and finite-difference step size belong in a numerical gradient-checking contract.

5. Equal reports, different updates

Let an SGD transition use the returned automatic gradient:

$$\theta_{t+1} = \text{fl}(\theta_t - \eta g(\theta_t)).$$

Starting at zero with $\eta = 1$, one update yields $\theta_1^H = -1$ and $\theta_1^Z = 0$. Both updates are exact small-integer binary32 operations. Both programs still return zero at their new states.

The attached script records the states after eight updates. At every recorded step $0 \leq t \leq 8$, the parameters are $\theta_t^H = -t$ and $\theta_t^Z = 0$, and both reported outputs are zero. All of these states stay within the proved zero-output interval. An observer retaining only that output trace therefore misses the parameter change.

This bounded trace demonstrates a particular observation gap. It supplies no result about indefinite execution, an end-to-end neural network, malicious behavior, or a practical attack. The scalar expression is an objective/readout for this example; we do not assert that it is a globally nonnegative loss.

6. Whole-expression compilation is a separate configuration

The rounding proof assumes separate materialization of the addition and subtraction. A compiler may rewrite the whole expression. JAX documents numerical changes caused by XLA algebraic simplification [4]; PyTorch also documents the importance of operation order and execution configuration for floating-point results [3].

We separately ran `jax.jit(jax.value_and_grad(H))` and the corresponding compiled `Z` at three inputs. These observations are recorded independently of the op-by-op witness:

Input	Compiled <code>H</code> output	Compiled <code>Z</code> output	Compiled gradients (<code>H</code> , <code>Z</code>)
-1	0	0	1, 0
0	0	0	1, 0
1	0	0	1, 0

In this measured JIT configuration, both compiled expressions still return zero at all three inputs, and their gradients remain one and zero. Thus this run supplies no example of JIT changing these forward values. It records an additional tested configuration; the interval proof still states its separate-rounding assumption explicitly.

The script measures compilation behavior instead of assuming that source parentheses enforce intermediate rounding. A different compiler, device, version, or optimization configuration needs its own check. We performed no GPU or `torch.compile` experiment.

7. A useful preservation contract

For this example, a forward-output comparison accepts replacing H by Z on the specified interval, while a comparison of automatic gradients rejects it. The replacement also changes an SGD transition. A transformation intended to preserve training should state which of these observations it preserves.

A simple sufficient contract for the displayed SGD recurrence is equality of the gradient function, the initial state, learning-rate sequence, dtype, and update evaluation order. Induction then gives equal parameter traces for the common execution domain. Stateful optimizers require additional state, including their moment buffers; randomized and scheduled computations require their generator and schedule state as well. Reference [1] develops the richer state and observation-preserving formulation.

For readers reviewing a compiler or training transformation, the practical questions are concrete: which forward values are compared, which reverse rules are retained, which optimizer state is preserved, and which numerical configuration is fixed? An output-only test answers only the first question.

8. Reproduction, relationship to prior work, and limits

This article is an expository companion to the already published paper [1]. Its distinct additions are executable PyTorch/JAX reproduction, a short reader-first derivation, two numerical controls, a bounded SGD trace, an independently measured JAX JIT configuration, and the three-panel figure. The original paper retains the broader localization, observation-fiber, and policy-descent arguments.

The source bundle contains `minimal_torch.py`, `minimal_jax.py`, `verify_frameworks.py`, pinned requirements, a full dependency lock, `framework-results.json`, the figure sources, and Markdown, LaTeX, PDF, and offline HTML reading editions. The framework report records every check and the actual

runtime versions. The source archive and verification script from [1] were checksum-identified and its original finite checks were rerun successfully; `provenance.json` records that separate check.

The evidence is for the recorded scalar CPU configurations. The finite input checks do not establish every floating-point input, hardware target, framework version, optimizer, or compilation mode. The elementary propositions have written proofs; no new machine-checked proof or security guarantee is claimed. Reproducing the artifact reruns the numerical and framework checks, not an independent referee assessment.

Automation disclosure. An AI assistant helped with exposition, Python implementation, framework execution, figure generation, reference checking, and packaging. The submitting owner authorized this Anonymous companion submission. The manuscript is prepared for editorial review.

References

- [1] Anonymous. *Homotopy Fibers and Observation-Preserving Semantics for Finite-Precision Training*. Higher Categorical Semantics, 2026. [Published article](#).
- [2] PyTorch contributors. *Autograd mechanics*. Official documentation, consulted 22 October 2026. [Documentation](#).
- [3] PyTorch contributors. *Numerical accuracy*. Official documentation, consulted 22 October 2026. [Documentation](#).
- [4] JAX authors. *Frequently asked questions: jit changes the exact numerics of outputs*. Official documentation, consulted 22 October 2026. [Documentation](#).