
BITWISE-IDENTICAL MACHINE LEARNING ACROSS METAL, CUDA, AND HIP

Andrew Hendel¹

ABSTRACT

The same machine-learning workload can produce different bits on different GPUs, changing predictions, learned models and subsequent training updates. Mojolearn is a GPU machine-learning library whose default *identical* mode produces bitwise-identical results across verified Apple, NVIDIA and AMD GPUs and x86-64 and Arm CPUs. Results agree bit for bit, not merely within a numerical tolerance. It supports training and inference for 57 machine-learning algorithms in 12 families, from decision trees to neural networks. Neural models trained in other frameworks can be served with identical outputs across vendors, and training workloads can be handed off between GPU vendors mid-run, or shared by GPUs of different vendors at once, while maintaining bitwise identity. Neural training and inference run in full FP32 floating-point arithmetic, without reducing the computation to integers. To my knowledge, I am the first to enable these capabilities.

One Mojo codebase implements these algorithms across Metal, CUDA and HIP under a shared numerical contract. Mojo compiles code for different GPUs, but maintaining that contract and implementing the algorithms is original work.

1 INTRODUCTION

A model may serve predictions on NVIDIA GPUs, run locally on an Apple laptop, and later be evaluated on an AMD server. Even with identical weights and inputs, different GPU implementations can produce different output bits. For deployments that require exact replay, hardware becomes another variable in serving, regression testing and auditing.

Mojolearn is a GPU machine-learning library whose default identical mode produces bitwise-identical outputs across verified devices and extends that guarantee to supported training computations. It covers 57 algorithms in 12 families, from decision trees to neural networks, with Apple M4, NVIDIA H100 and AMD MI325X GPUs and x86-64 and Arm CPUs in its verification scope. Unsupported dtypes, shapes or options raise an error rather than silently weakening the numerical guarantee. Section 3 states the conditions and limits of that guarantee.

Inference: any model, the same bits on every vendor. A model does not need to be trained in Mojolearn to be served by it. Transformer and Mamba blocks load weights trained in other frameworks, and the same weights and inputs then return the same output bits on Apple, NVIDIA and AMD

GPUs and on CPUs. A team can test a deployed model on different hardware, replay a recorded decision during an audit, and replace serving hardware without numerical change. The guarantee covers the model once it runs in Mojolearn. The framework that trained it rounds differently, and its own outputs vary from GPU to GPU, so Mojolearn’s outputs can differ from that framework’s in the last bits.

Training: the same model on every vendor. Training in Mojolearn is also bitwise identical across vendors. Given the same code revision, data, hyperparameters and seed, Apple, NVIDIA and AMD GPUs produce the same model state, gradients, optimizer state and checkpoint bytes. A checkpoint written on one vendor can therefore resume on another and follow the same trajectory as uninterrupted training. An identical model gives identical answers only while it is also served in identical mode: if the trained weights are served by another framework, that framework’s arithmetic applies and outputs can again differ from GPU to GPU.

Portability is inherited; identity is not. Mojo and MAX provide compilation and execution across Metal, CUDA and HIP (Modular, 2026a; Lattner et al., 2021); they do not make the arithmetic agree across those backends. Mojolearn defines the shared numerical rules and implements the algorithms that obey them. Its own numerical kernels control arithmetic and reduction order, replacing vendor math routines where their results would depend on the backend. The

¹Latent Archon LLC, United States; ORCID 0009-0000-9877-3623. Correspondence to: Andrew Hendel <ajhendel@latentarchon.com>.

contribution is the numerical contract and its implementation across the library’s algorithms; Mojo and MAX supply the execution substrate. Section 5 explains how the contract is enforced.

Bitwise identity matters throughout the library, beyond neural networks. In a tree learner, rounding can change the winning split and every subsequent node; in k -means, it can move a point to another cluster and change every later centroid. These are applications of one numerical contract across many algorithms.

Contributions.

1. *Bitwise-identical machine-learning algorithms*: a single Mojo codebase supports training and inference for 57 algorithms in 12 families, from decision trees to neural networks.
2. *Cross-vendor handoff and mixed-vendor training*: a training run can move from one GPU vendor to another mid-run, or be shared by Apple, NVIDIA and AMD GPUs working on one model at once. Either way, the training state after every step is bitwise identical to a single GPU’s (Sections 7.2 and 7.5).
3. *A numerical contract that separates arithmetic from scheduling*: the contract fixes the operations that determine model bits, including reduction order, rounding and elementary functions. Tiling, thread layout, memory placement and kernel scheduling remain free, so each backend can be tuned for its own hardware without changing a bit (Section 5). This vendor-specific tuning has only begun.
4. *Apple M-series GPU support*: every algorithm runs on the GPUs of Apple M-series Macs, where XGBoost, LightGBM, CatBoost and cuML have no supported GPU path and scikit-learn reaches the GPU for only a few estimators (Section 9).

To my knowledge, these are the first cross-vendor bitwise-identical GPU training and inference implementations. I achieve this in full FP32 floating-point arithmetic for neural networks, without reducing the computation to integers. Prior bitwise results are single-vendor or emulate one vendor’s arithmetic (Section 9).

Building a cross-vendor numerical profile required an extensive implementation effort. Mojo 1.0 was released on August 11, 2026; development of Mojolearn began on August 19, 2026, and the original preprint was submitted to arXiv on September 8, 2026. Release 0.7.0, archived on September 9, 2026 (Hendel, 2026b), documents three-vendor training identity for a byte-level language model. As of September 20, 2026, Mojolearn comprises 429,742 lines of original Mojo code: 176,999 lines of library implementation and 252,743 lines of verification fixtures, test harnesses and

benchmarks, with a further 193,171 lines of Python, shell and other code (Appendix C). Its public repository holds 5,316 commits made between August 19 and September 20, 2026. The system was built with AI coding assistants under the author’s direction, and every result reported here is checked by the verifier, not taken from the code’s authorship. This scale is necessary to cover the 57 algorithms evaluated in this work.

2 THE SYSTEM

Users call Mojolearn from Python through scikit-learn-shaped estimators (Pedregosa et al., 2011) and neural blocks. One Mojo source compiles for Metal, CUDA and HIP and ships as an Apple silicon wheel and a Linux x86-64 wheel. In identical mode, all numerical work, including the portable elementary functions, runs in the library’s own compiled Mojo kernels (Modular, 2026a), without calling MAX’s matrix-multiplication kernels, vendor BLAS and solver libraries, or external ML frameworks.

The user-facing contract. One flag, `numeric_mode="identical"`, selects the default numerical contract on every supported interface. For example, a caller passes weights to a transformer block and runs its forward pass in identical mode. A fresh process using the same input and weight bytes, hyperparameters, seed and numerical profile returns the same bits on a verified device.

Table 1 summarizes the public API, from neural inference and training to classical machine learning. Transformer and Mamba blocks expose forward computation and stateful decoding. Fixed-shape training interfaces support a small MLP and a byte-level language model; Section 7.2 reports their bitwise-identity coverage. The tree family includes symmetric, depth-wise and loss-guided gradient boosting, plus random-forest and extremely-randomized-tree classifiers and regressors (Breiman, 2001; Geurts et al., 2006). Other interfaces cover clustering, neighbors, linear and kernel models, decomposition, manifold embedding, time series and statistical resampling. Every interface also runs on x86-64 and Arm CPUs, and the library-wide verification covers this CPU reference build alongside the three GPU vendors.

Table 1 counts the algorithms. The breadth matters as evidence of generality. Table 2 shows why: each family exposes a different way for hardware-dependent arithmetic to alter model state or outputs, and the same user-visible mode must close all of them.

The implementations build on published algorithmic designs: CatBoost’s oblivious-tree GPU design for symmetric boosting and its depth-wise and loss-guided trees (Prokhorenkova et al., 2018); fixed-point gradient histograms from the XGBoost and LightGBM GPU learners (Chen & Guestrin, 2016; Ke et al., 2017) and Light-

family	algorithms	count
gradient boosting	symmetric-tree, depth-wise and loss-guided boosting, ordered boosting	4
forests	random forest, extremely randomized trees, isolation forest	3
clustering	k -means, DBSCAN, HDBSCAN, agglomerative, spectral, Gaussian mixture	6
neighbors, density and search	brute-force nearest neighbors, k -NN prediction, radius neighbors, random ball cover, IVF-Flat, kernel density	6
linear models	least squares, ridge, lasso, elastic net, logistic regression	5
kernel methods and Gaussian processes	support vector machines, kernel ridge, Gaussian process regression, Gaussian process classification, Nyström, random Fourier features	6
decomposition and manifold	PCA, truncated SVD, UMAP	3
time series	ARIMA, Holt-Winters exponential smoothing, KPSS test	3
preprocessing and resampling	standard scaler, min-max scaler, bootstrap, permutation test, Monte Carlo integration	5
neural blocks and optimizers	transformer, Mamba-1, Mamba-2, Mamba-3, Samba hybrid stack, MLP, Adam, AdamW, SGD with momentum	9
language model and tokenizer	decoder language model, byte-level BPE tokenizer	2
linear algebra	matrix product, Cholesky, QR, symmetric eigendecomposition, singular values	5
total		57
also provided, not counted	24 evaluation metrics; cross-validation; CPU inference for saved models; stepwise decoding with explicit state; checkpoint-loaded causal language models; BF16 and INT8 weight storage; tokenized corpora; two-GPU execution plans; the verifier	

Table 1. The 57 algorithms by family. A classifier and a regressor of the same method count once, and losses, kernels, distance metrics, penalties, starting rules and two-GPU execution are configurations of an algorithm, not algorithms. Neural configurations are summarized in Table 11; training coverage is described in Section 7.2.

family	what can vary	effect on the model	identical-mode response
neural building blocks	matrix products, reductions and state updates	different activations, gradients or training state	portable primitives and explicit rounding barriers
trees and forests	histogram order, split-score rounding and ties	different split, partition or tree structure	fixed accumulation, frozen geometry and total tie rules
clustering and neighbors	distance reductions, square root, traversal and output order	different neighbors, memberships, clusters or labels	portable distance arithmetic and canonical ordering
linear and decomposition	matrix-product order, solver folds and closed vendor libraries	different coefficients, margins or components	portable matrix operations and fixed fold order
time series	recurrences, elementary functions and pivot ties	different likelihoods, parameters or forecasts	pinned functions, pivots and decision rules

Table 2. Algorithmic breadth exercises distinct numerical failure modes. The claim is one contract across these mechanisms, not novelty in the learning algorithms themselves.

GBM’s feature sampling; the batched-level design used by cuML for forests; and the corresponding cuML, RAFT and cuVS designs for classical methods (RAPIDS Development Team, 2026). This work does not claim new learning algorithms. Its contribution is a numerical contract that a useful collection of them share, and the kernels and numbered deviations that make it hold.

Beyond estimators. Saved classical models support CPU inference, and checkpoint loading and assembly support language-model decoding from Hugging Face-format weights. The library also provides a byte-level tokenizer and tokenized corpora. Neural blocks expand BF16 and INT8 weight storage to FP32; a separate INT8 matrix profile uses exact integer accumulation (Section 5). Explicit execution plans distribute supported workloads across GPUs while

preserving the prescribed reduction order (Section 7.5). The installed verifier checks the identity contract (Section 6).

Numeric modes. Identical is the default and, outside the tree learners, the only mode. Gradient boosting, random forests and extremely randomized trees also offer *deterministic* mode, which repeats bits on one device but not across vendors, and *fast* mode, which promises only speed. Table 7 summarizes the three modes, and Figure 1 shows one computation that repeats on each device yet differs across vendors.

3 SCOPE AND LIMITATIONS

The guarantees below require the same code revision, numerical profile, input bytes, initial state, hyperparameters and seed, with identical mode selected and the configuration

covered by the verification record.

3.1 Scope of inference

Every verified device returns the same output bits for the supplied model weights, whether those weights were trained in Mojolearn or elsewhere. Inference is also batch invariant (He & Thinking Machines Lab, 2025): a sequence’s output bits do not depend on how many other sequences share the batch.

3.2 Scope of training

Every verified device produces the same learned model and supported training state, including gradients, optimizer state and checkpoint bytes. A refused configuration never counts as a pass.

3.3 Limitations

These limits apply throughout; the rest of the paper does not repeat them.

- *Devices.* The record was collected on an Apple M4, on NVIDIA H100, A100 and RTX 4090 GPUs, on AMD MI325X and MI300X GPUs, and on AMD EPYC and Apple M4 CPUs (Appendix A). Mojo supports GPU programming on Apple M-series GPUs and substantially all NVIDIA and AMD GPUs (Modular, 2026b). I believe Mojolearn produces bitwise-identical results across all of them, and the record already spans three NVIDIA and two AMD GPU models, but a GPU is not covered until it passes the same checks (Table 3).
- *Precision.* The guarantee is for FP32 arithmetic. FP64 input is converted or refused.
- *What was tested.* The record covers the configurations, inputs, model shapes and release it names. A call that succeeds on an untested configuration is not a verified result.
- *Same numerical rules.* Replay must also run in identical mode. A result from fast mode or another framework is not covered, and weights trained elsewhere reproduce under Mojolearn’s rules, not their original framework’s bits.
- *Option coverage.* Some interfaces accept more options than the record covers (Appendix A.4).
- *Input boundary.* The guarantee begins at the estimator input; preprocessing performed outside the library is not covered.
- *Quality.* Identical bytes can encode a poor model, so quality validation remains a separate obligation.
- *Speed and cost.* Performance is not evaluated here. Mojolearn is five weeks old and has not received the years of tuning that established libraries have, so its current

speed reflects the maturity of the implementation as well as the cost of identity (Section 8).

Table 3 also lists prospective Qualcomm Adreno and Intel Xe hardware against the contract’s requirements. Each new backend must pass verification; hardware that violates a pinned assumption is refused. Changes to the numerical rules require a new profile version.

4 WHY GPUS PRODUCE DIFFERENT MODELS

GPU portability does not fix the numerical choices made during execution. Two backends can evaluate the same expression using different intermediate precision, reduction grouping or elementary-function implementations. The result can differ in its last bits even when the model, inputs and seed are identical. Neural networks expose these differences during both inference and training; tree learners also turn them into discrete structural changes.

Neural inference: the same weights can produce different outputs. A transformer composes matrix products, attention, normalization, activation functions and residual updates. A dot product is a sum of products: changing the precision of those products or how partial sums are grouped can change its result. Attention and normalization add reductions, division, exponentials and square roots. Agreement in the matrix products alone is therefore insufficient; each subsequent operation must preserve the same numerical rules. A difference in an intermediate activation can propagate through later layers to the output logits. If it changes the ordering of nearly tied logits, even greedy decoding can select a different token and therefore a different input for the next decoding step.

Mamba blocks introduce another dependency through recurrent state. Each state update contributes to later outputs, and stepwise decoding carries that state between calls. Matching the weights and current input is insufficient if the saved state already differs. Cross-vendor identity must cover both outputs and the state that subsequent computation consumes.

Neural training: numerical differences become model differences. Backward computation introduces further matrix products and gradient reductions. Gradient clipping and optimizer updates then combine those gradients with parameters and accumulated state. In AdamW, differences can enter the first and second moments as well as the updated parameters, which are inputs to the next training step. Fixing the random seed does not fix these arithmetic choices. Nor does a kernel that repeats on one GPU necessarily agree with a different vendor’s implementation.

Exact training replay therefore requires a common numerical profile through forward computation, loss, backward computation and optimizer updates. Checkpoint continu-

	verified, measured			buildable, unvalidated	prospective, not buildable	
	Apple M4	NVIDIA H100	AMD MI325X (CDNA)	AMD RDNA	Qualcomm Adreno	Intel Xe
square root	correctly rounded	approximate	correctly rounded	—	—	—
denormals	flushed	kept	kept	—	—	—
$a \cdot b + c$	fused	fused	fused	—	—	—
$\max(+0, -0)$	-0.0	+0.0	+0.0	—	—	—
NaN payload	0x7fc00000	0x7fffffff	0xffc00000	—	—	—
lockstep lane width	32	32	64	32	8, compiler-chosen	8, compiler-chosen
vendor FP32 matrix product	FP32	TF32 by default	FP32	FP32	FP32	FP32
float atomics	global only	yes	yes	yes	no	yes
threadgroup memory per block	32 KB	48 KB	64 KB	64 KB	32 KB	64 KB
meets minimum hardware requirements	yes	yes	yes	yes	yes	yes

Table 3. Current GPU coverage and requirements for future backends. Arithmetic results are measured on Apple M4, NVIDIA H100 and AMD MI325X. AMD RDNA is buildable but unvalidated; Adreno and Xe entries describe prospective targets from vendor documentation, unsupported by the evaluated toolchain. The minimum hardware requirements are 32 KB of threadgroup memory, threadgroup integer atomics and 1024-thread blocks. Meeting these requirements is only a starting point: each new backend must pass the same verification. Identical mode supplies shared arithmetic where hardware behavior differs, including square root, denormal handling, matrix products and a logical reduction width of 32 lanes.

ation additionally requires preserving the state needed by the next step, including parameters, optimizer moments and counters, together with the same subsequent input sequence. These dependencies explain why the neural results compare intermediate outputs and training state as well as final checkpoint bytes (Section 7.2). A bitwise difference need not cause a material change in model quality to matter for exact replay: it already changes the artifact being compared.

Tree training: numerical differences can change model structure. Tree training turns numerical differences into structural decisions. CatBoost’s GPU histogram implementation uses floating-point atomics, so thread arrival determines the summation order. Its partition-statistics kernels derive the chunk count from the number of streaming multi-processors: a setting that looks like scheduling determines how the sum is grouped (CatBoost Developers, 2026a). Both choices can change the totals and therefore split scores. Near a tie, that difference can select another split, alter the partition and change subsequent trees. Border selection, leaf updates and tie-breaking must therefore follow the same numerical rules as histogram accumulation.

Table 2 identifies the corresponding hazards in the other algorithm families. Section 5 explains how the shared numerical contract controls them.

5 HOW IDENTICAL MODE WORKS

Identical mode controls operations that can change model bits, including floating-point accumulation, elementary functions, tie-breaking and reduction settings derived from the device. For each operation, Mojolearn fixes the relevant setting, uses a portable implementation, or rejects an unsupported configuration with an explanatory error. Operations that do not change arithmetic order or precision remain free to use each backend’s normal scheduling. Table 4 summarizes these choices; Appendix B.2 lists every place they

what could cause disagreement	what identical mode does
hardware chooses a setting that affects the result	use the same setting on every backend (pin)
GPU implementations calculate differently	use the library’s portable implementation (replace)
no conforming implementation is available	stop with an explanatory error (refuse)

Table 4. The three ways identical mode prevents silent cross-GPU differences.

apply (Table 12).

Machine parameters can change the result. A block count, a replication factor and a reduction width all determine which partial sums are combined. They are therefore numerical choices, even when an implementation labels them as scheduling or tuning parameters. This is established in the reproducible-summation literature (Demmel & Nguyen, 2013; Demmel et al., 2016; Collange et al., 2015), appears as a product constraint in vendor math libraries (Lubin & Mueller-Albrecht, 2024), and is restated for inference as batch invariance (He & Thinking Machines Lab, 2025). Identical mode pins every machine-derived parameter that reaches floating-point arithmetic.

Floating-point and integer matrix units differ. The FP32 profile avoids vendor floating-point matrix units: their fragment shapes and internal accumulation do not implement the library’s specified rounding order, and TF32 additionally rounds operands before multiplication. RepDL identifies hardware-specific low-precision matrix arithmetic as a similar obstacle (Xie et al., 2025). My measured FP32 workloads therefore retain the cost of the general-purpose arithmetic path. The separate INT8 profile admits integer

matrix units because its bounded Int32 products and sums are exact before the pinned conversion to floating point. Changing the grouping of an overflow-free integer sum cannot change its value. The FP32 workloads do not use those units.

Ordinary-looking choices change bits. DBSCAN normally sizes its batches from free device memory, so unrelated memory use could change the graph decomposition; identical mode pins the batch size. A GPU’s default square root may be an approximate instruction, so identical mode uses its own implementation on every backend. Table 3 summarizes the measured arithmetic differences that portable replacements must cover.

Worked example: a neural matrix product. The FP32 matrix-product profile used by transformer projections separates the logical arithmetic from its physical execution. For one output $C_{ij} = \sum_{p=0}^{K-1} A_{ip}B_{pj}$, partition the reduction into contiguous leaves of length L . For $0 < K \leq 128$, $L = K$; otherwise $L = \max(128, \lceil K/1024 \rceil)$. These constants belong to the numerical profile, so neither batch size nor GPU occupancy can change them. Within each leaf, start at positive zero and visit p in ascending order:

$$a_{p+1} = F(\text{fma}_{32}(F(A_{ip}), F(B_{pj}), a_p)).$$

Here F flushes a subnormal to zero of the same sign, and fma_{32} rounds the fused product and sum once to FP32. All FP32 rounding in these examples uses round-to-nearest, ties-to-even; fl_{32} denotes that rounding operation. Flushing after every update matters: flushing only the final partial would allow a backend that retains subnormals to carry a different intermediate into the next update.

Combine adjacent leaf partials in a fixed binary tree, using $F(F(x) + F(y))$ with FP32 addition at every node. Carry an unpaired final partial unchanged to the next level; do not pad it with zero. For example, $K = 300$ gives leaves of 128, 128 and 44 products, and the fold is $F(F(s_0 + s_1) + s_2)$. An empty reduction returns positive zero. The input loads, leaf stores, fold reads and final output also obey F . GPU threads may compute independent leaves or output tiles concurrently, and intermediate values may live in registers, shared memory or a separate buffer, but each output must follow this same dependency tree. A GPU whose hardware runs 64 threads in lockstep, where the others run 32, cannot substitute its native reduction for the prescribed pairing. Thus changing tile size or batching changes the schedule without changing the arithmetic; changing a leaf boundary, pairing or rounding step requires a new numerical profile.

Worked example: a tree histogram and split. *Scale and bound.* The fixed-point histogram path must first turn each floating-point statistic into the same integer on every device. Let N be the row count and M the larger of two sums over

all rows, the absolute gradients and the absolute weights or Hessians, each reduced in a fixed order. The scale is a power of two chosen to fit the accumulator budget. In the row-count-aware branch, for $0 < N \leq 2^{30} - 2^{28}$, the budget is $B = 2^{30} - 1 - N$ and the scale is the largest $s = 2^e$ satisfying $Ms \leq B$; an all-zero input uses $s = 1$. This range keeps the budget at or above the selector’s $2^{28} - 1$ floor. The row allowance accounts for up to one quantization unit per row. For example, $N = M = 10^6$ selects $s = 1024$: every subset sum is bounded in magnitude by $Ms + N = 1.025 \times 10^9 < 2^{30}$, within signed Int32. Bounding absolute values, rather than the final signed sum, also bounds intermediate sums when positive and negative gradients cancel.

Quantization and accumulation. The histogram quantizer uses deterministic dither to reduce the systematic bias that truncation or repeated rounding of equal statistics can introduce. A fixed row hash varies the rounding decisions reproducibly. For a statistic v , compute $x = F(F(v)s)$ and $b = \lfloor x \rfloor$, then form the integer $q = b + 1[\text{fl}_{32}(F(x - b) + u) \geq 1]$, where $u \in [0, 1)$ comes from that integer hash of the row key. The key and hash are independent of thread arrival order. Separating the integral and fractional parts avoids rounding the full $x + u$ before taking its floor. Integer atomics can then accumulate these bounded values in any arrival order. At the conversion boundary, an integer total Q becomes $F(\text{fl}_{32}(\text{fl}_{32}(Q)/s))$; converting partials early and summing them as floats would lose this order independence.

Split selection. Identical histograms are necessary but insufficient for identical trees. Partition totals and split-score expressions retain their pinned folds and rounding rules. The final comparison orders finite candidates by ascending negated gain, then unsigned feature index, then unsigned bin index. A full tie keeps the later candidate, and the traversal order is fixed, so the choice is the same on every device. These rules carry agreement from histogram values into the discrete split that determines the next partition. Fixed-point quantization changes the numerical computation; its resolution and the resulting model quality remain separate from the identity guarantee. Together, the examples illustrate the scheduling rule: prescribe the order where rounding matters, and permit order changes where bounded integer arithmetic is exact. Appendix B.2 places these mechanisms in the library-wide numerical contract.

From identical updates to training handoff. The supported language-model trainer checkpoints parameters, optimizer moments, initialization flags, the completed-step counter and optimizer settings as raw state. Companion metadata binds that checkpoint to the numerical profile, code revision, corpus and token schedule, including the position at which training resumes. The receiving device restores the same state and consumes the same next batch with the same logical microbatch partition. Forward computation,

loss, backward computation and the optimizer update follow the shared numerical rules, so the next state matches the uninterrupted run. Repeating this argument gives the same subsequent trajectory for supported computations. Transferring weights alone would not suffice: different moments or a different step counter can change the next update even when the forward outputs agree.

6 REPRODUCIBILITY AND VERIFICATION

Anyone can check the claim. The installed package ships a verifier with three complementary checks: comparison with recorded references, local GPU–CPU inference comparison, and comparison between independent runs.

- *Against recorded references.* The verifier trains each covered algorithm on fixed fixtures and compares hashes of training state, held-out predictions and saved-model bytes with references recorded on Apple, NVIDIA, AMD and CPU. A match means the user’s machine joins those columns. Input hashes are checked first, so a changed fixture is not mistaken for different arithmetic.
- *GPU against CPU on the user’s own machine.* Each model is fitted on the user’s GPU, and the same held-out predictions are then requested from the GPU and from the saved model reloaded on the CPU. The user produces both sides on two different pieces of hardware, so the check trusts no recorded reference, and it runs on any supported GPU, including a laptop’s, not only the recorded devices. This checks inference from the GPU-fitted model; it does not independently repeat training on the CPU. It also checks batch invariance: a row keeps its output bits when its batch neighbors change.
- *Two independent parties.* Two people run the verifier on their own machines and compare the resulting evidence documents, with no GPU needed for the comparison and no involvement from the author. Each can first publish a commitment to their document, so neither can copy the other’s results after seeing them. The comparison reports agreement only after ruling out mismatches, self-contradicting results, uncomputed cells and a document compared with itself.

These checks establish cross-device consistency while sharing the library’s implementation; algorithmic correctness and model quality require separate checks against independent references and quality criteria. Every check reports agreement, divergence, a missing reference, refusal or non-applicability, and only agreement counts toward a verified verdict. A self-test must accept an unchanged fixture and detect one whose input differs only in the last bit. For stateful sequence models, checks also compare token-by-token decoding against a full forward pass. Transformer

and Mamba checks span batch sizes from 1 to 1,024 and sequence lengths on either side of every kernel’s chunk boundary, where the kernels choose different tilings. Training results are compared at every recorded training stage, not only through predictions, because two different models can agree on a test set. A coverage inventory maps the 57 algorithms and 273 verifier configurations to their checks, and evidence documents carry binary digests and provenance (Appendix A).

Optional trace hashing. Verification and debugging builds can enable a *trace card*: an ordered list of stage names and hashes of raw buffer bits, inputs and host scalars that affect training decisions. A single model hash reports only that two runs differ; comparing cards identifies the first recorded stage that differs, and optional buffer dumps expose per-cell bit patterns and how far apart the values are. Normal training omits this instrumentation. On an unpinned k-NN computation, for example, the card isolates CUDA’s approximate square root: the NVIDIA distance stage differs in the last bit while the preceding norms and resulting neighbor indices agree (Figure 1). Identical mode therefore uses Mojolearn’s pinned square root wherever normalization affects model bits.

7 RESULTS: THE SAME MODEL ACROSS THREE VENDORS

The verification record for release 0.8.10 combines independently recorded Apple Metal, NVIDIA CUDA, AMD HIP and CPU results. Across 214 single-device configurations and nine fixtures, all 7,452 applicable numerical parts agree bit for bit across the CPU and all three GPU vendors, with no mismatches or missing numerical vendor comparisons. Another 18 saved-model parts agree across all three GPUs; the CPU loads those models rather than writing them. The fixed evidence archive contains the reference table, its underlying records and the machine-readable coverage audit (Appendix A).

A *configuration* is one algorithm with one set of options. A *cell* is one configuration on one fixture; its *parts* compare training state, saved-model bytes, held-out predictions and applicable batch or sequence properties. The single-device record contains 1,926 cells. Table 5 gives their coverage by family, and Appendix A gives the counts by part and fixture. The reference table also contains 59 two-GPU configurations, reported separately in Section 7.5; they are not included in these totals.

7.1 What a cell compares

Each configuration compares the numerical parts its interface supports: training state, saved-model bytes and predictions on held-out rows. Stateless operations and interfaces without serialization declare the corresponding parts

#	stage	type	n	Apple M4	NVIDIA H100	AMD MI325X
0	knn.index_norm	f32	20000	74a472e4925ec44e	74a472e4925ec44e	74a472e4925ec44e
1	knn.query_norm	f32	2000	18da124518cd0e52	18da124518cd0e52	18da124518cd0e52
2	knn.out_dist	f32	20000	18499d126dfd6ffe	563af8012606fe61	18499d126dfd6ffe ◀
3	knn.out_idx	u32	20000	379df04034c318b9	379df04034c318b9	379df04034c318b9
4	knn.sorted_dist	f32	20000	18499d126dfd6ffe	563af8012606fe61	18499d126dfd6ffe ◀
5	knn.sorted_idx	u32	20000	379df04034c318b9	379df04034c318b9	379df04034c318b9

Figure 1. Optional trace hashing identifies the first divergent operation in an unpinned k-NN computation. The standard-library path first differs on NVIDIA at the distance buffer (◀), and that difference survives in its sorted copy; the preceding norms and resulting neighbor indices agree. This is CUDA’s approximate PTX square root. Identical mode uses Mojolearn’s pinned square root and produces matching bits without requiring trace hashing during normal training.

record group	configs	cells	parts	parts with equal bits		divergent
				three GPU vendors	and CPU	
gradient boosting	31	279	1,089		1,089	0
forests and isolation forest	13	117	459		459	0
clustering and mixtures	19	171	639		639	0
neighbors, indexes and density	25	225	909		909	0
linear models and SVM	24	216	837		837	0
kernel methods and Gaussian processes	20	180	702		702	0
decomposition and UMAP	5	45	189		189	0
time series	9	81	288		288	0
neural blocks	11	99	684		684	0
language-model training and inference	8	72	306		306	0
reduced-precision weights and matrix products	15	135	684		684	0
optimizers, losses, embeddings and tokenizers	13	117	261		261	0
linear algebra	6	54	99		99	0
metrics, preprocessing and model selection	15	135	324		324	0
total	214	1,926	7,470		7,470	0

Table 5. Single-device verification record for release 0.8.10. A part is one compared quantity of one cell, such as training state, saved-model bytes or held-out predictions. Every part has equal recorded bits across the Apple, NVIDIA and AMD columns, and all but 18 also across the CPU column; in those 18 saved-model comparisons the CPU loads the GPU-written file instead of writing its own (Table 6). The NVIDIA column is drawn from H100, RTX 4090 and A100 records and the AMD column from MI325X and MI300X records (Appendix A). The rows group configurations as the verifier records them, which splits some of the 12 algorithm families of Table 1. Counts exclude non-applicability markers and the 59 two-GPU configurations.

non-applicable. The saved file is loaded back and must predict what the model in memory predicted. Configurations whose interface allows it are also held to properties that a single hash cannot show. A row must keep its output bits when its batch neighbors change, at small batches and at a serving-scale batch of 1,024 rows. A padded or ragged batch must give each sequence the bits it gets when run alone. Token-by-token decoding must equal the full forward pass. Per-row gradients must remain batch invariant; accumulated gradients must agree for splits that preserve the prescribed reduction tree. The log-probability a sampler records must equal the one a trainer recomputes. Table 6 in the appendix counts the configurations and cells under each property.

7.2 Bitwise-identical neural inference and training

The neural configurations apply the same contract to forward computation and to training. Transformer, Mamba-1, Mamba-2, Mamba-3 and hybrid blocks match across Apple, NVIDIA and AMD in outputs, recurrent state and gradients. Training configurations run embedding, forward and backward passes, cross-entropy loss and AdamW with gradient clipping, and the three vendors write byte-identical checkpoint files without consuming another run’s rounding log. The contract therefore composes through gradient computation, optimizer updates and checkpoint serialization.

The record includes full and windowed transformers, the three Mamba variants, hybrid stacks, MLPs, language-model training and reduced-precision weight storage. Table 11 reports configurations by group; Table 6 separates

forward, saved-model, gradient and sequence properties so that a match on one property is not presented as coverage of another.

A separate byte-level language-model demonstration trains the same model independently, from initialization to the final step, on each of the three vendors, and compares every recorded array of every training step. No run replays another vendor’s recorded steps. A checkpoint resumes on another vendor and rejoins the uninterrupted trajectory in both directions, and CPU replay reproduces individual recorded training steps. Appendix A.5 describes this demonstration separately from the fixture record.

7.3 Trees, classical models and the full library

The same contract holds beyond neural networks, for gradient boosting, forests, k -means, DBSCAN, PCA, linear and kernel models, ARIMA and the other classical families in Table 5. Each configuration is fitted on nine fixtures, a base case and adversarial cases built from many-way ties, unstructured hashed values, column magnitudes spanning eight orders, subnormal values, duplicated and constant columns, sizes that no block divides and all-negative inputs (Table 8). Models trained on one vendor load and produce bitwise-identical predictions on the others. A configuration the contract cannot cover is refused by name instead of being approximated, and such a refusal is recorded as a refusal, never as a match.

7.4 The verifier can fail

A table of matching hashes is only evidence if a wrong implementation would have produced a mismatch. Each family therefore has negative controls, builds of the library with one deliberate arithmetic change such as a reversed fold, a transposed operand or a dropped flush to zero. A control counts only when a committed pair of runs shows that it moved the hash of that configuration. A control that exists in the source but has not been seen to move a configuration counts for nothing. By that rule, a changed build has been observed to move the bits of 205 of the 214 single-device configurations; the other nine have a declared control that has not yet been observed to move them (Appendix A.3). The verifier’s own self-test also detects an input that differs only in its last bit.

7.5 Two-GPU execution

Changing the device count can change the answer when it changes how sums are grouped. Mojolearn instead divides work into pieces that each GPU computes exactly as a single GPU would, such as whole trees in a forest or whole microbatches in neural training, and combines them in a fixed order. The reference table contains 59 two-GPU configurations covering classical and neural workloads. All 1,935 of their numerical parts have equal bits across the Apple,

NVIDIA and AMD columns, with no missing value and no disagreement, and the 771 parts that have a numerical CPU role match the CPU as well (Appendix A.4).

Two-GPU execution is also checked on two physical GPUs. The shipped verifier runs each configuration once on one GPU and once across two, confirms by device identifier that two distinct GPUs were used, and fails if a single bit moves. With release 0.8.10 installed from PyPI on two NVIDIA RTX 4090 GPUs, the verifier’s self-test passes, its quick check of one configuration per family is verified, and a sweep of all nine fixtures compared 1,935 parts between one and two GPUs: all are identical and none diverges. On two AMD MI300X GPUs, two-GPU records equal the same reference values. More than two devices in one machine and multi-GPU throughput are not measured. The same fixed fold also lets an Apple, an NVIDIA and an AMD GPU in separate machines train one model together with identical state after every step (Appendix A.5). An Apple M4, an NVIDIA H100 and an AMD MI300X trained one small decoder for 6 steps: the three workers reported the same state hash after every step, equal to a single GPU’s, and all 1,512 assignments of the recorded microbatch gradients to vendors reproduce the same sum.

8 WHAT IDENTITY COSTS

Cross-vendor bitwise identity carries implementation and execution costs: identical mode gives up vendor matrix units, approximate elementary functions and device-chosen reduction geometry. These constraints require additional implementation work and can increase execution time. The present evaluation establishes bitwise identity; it does not quantify performance against competing implementations.

9 RELATED WORK

Reproducible arithmetic. That a block count, processor count or partition is a summation order is well established (Demmel & Nguyen, 2013; Demmel et al., 2016; Lubin & Mueller-Albrecht, 2024; He & Thinking Machines Lab, 2025), and integer gradient histograms are established in tree boosting (Shi et al., 2022); Mojolearn uses fixed-point accumulation in the tree histogram because a bounded integer sum does not depend on its order. ReproBLAS (Demmel et al., 2016) and Intel MKL’s conditional reproducibility mode (Intel, 2024) ship reproducible summation and linear algebra. Climate models have shown bitwise identity across compiler versions and flags (Li et al., 2016), and molecular-dynamics codes document how results change with processor count (OpenMD, 2013). Platform differences in denormals, FMA contraction and elementary functions are documented (Dawson, 2013; Whitehead & Fit-Florea, 2011); my measurements confirm them rather than discover them.

Deterministic modes. Framework deterministic modes switch known-nondeterministic operations to deterministic ones or refuse them (PyTorch, 2026; TensorFlow, 2024), and NVIDIA’s primitive library grades its reductions as non-deterministic, run-to-run deterministic or GPU-to-GPU deterministic (Al Awar & Singanaboina, 2026). Other systems reproduce training in a fixed environment through deterministic settings, environment capture or replay (Chen et al., 2022; Heumos et al., 2023). These give repeatability on one platform, which is weaker than identity across vendors: Figure 1 shows a computation that repeats on each device and still differs between them.

Bitwise training. RepDL (Xie et al., 2025) follows the same strategy as this work, with correctly rounded FP32 operations, a fixed summation order and deliberate FMA use, and states the cross-system goal. Its repository implements CPU and CUDA backends with a reproducible training example; the report has no evaluation section and reports no hardware it was tested on. Optimistic Verifiable Training (Srivastava et al., 2024) trains exactly across three NVIDIA GPUs by having the trainer record rounding decisions that an auditor replays. Verde’s RepOps (Arun et al., 2025) fixes operation order inside CUDA operators across the NVIDIA GPU configurations it tests. Hawkeye (Badash et al., 2026) emulates NVIDIA Tensor Core arithmetic on a CPU. Their GPU implementations target NVIDIA, or their verification reproduces NVIDIA arithmetic on a CPU.

Concurrently with this work, OPEN-1B (Donaghy et al., 2026) trains a 1.6-billion-parameter language model on 48 NVIDIA H100 GPUs and publishes a state hash for every step, so that an auditor can replay any single training step on a CPU, an NVIDIA GPU or an Apple GPU and obtain the same bits. It reaches a second GPU vendor at a scale far beyond the fixtures used here. Its large matrix products run in INT8 and accumulate in Int32, where a sum of sufficient width is exact in any order (Chen, 2026a), and it also controls floating-point reductions, fusion, subnormals and optimizer arithmetic; Mojolearn’s neural workloads keep their matrix products in FP32. OPEN-1B does not cover AMD GPUs, trains on one vendor and replays on others rather than training on each, and covers one model architecture. OPEN-1B verifies individual steps by single-device replay against published per-step hashes, with many auditors intended to cover the run collectively; Mojolearn compares complete independent training runs on each vendor.

Bitwise inference. EigenAI (Alves et al., 2026) fixes the GPU architecture and reports that A100 and H100 outputs differ, and Cankaya (2026) reproduces NVIDIA inference through CPU emulation. Batch-invariant and parallelism-invariant kernels make repeated serving deterministic (He & Thinking Machines Lab, 2025; Zhang et al., 2025; vLLM Project, 2026), LLM-42 (Gond et al., 2026) reaches the

same goal with existing kernels through scheduling and a verify-and-rollback loop, and Gensyn’s REE packages reproducible operators with checkable receipts on NVIDIA GPUs and CPUs (Gensyn, 2026; 2025). Power-of-two INT8 scales make two quantized kernels agree on one GPU (Chen, 2026a;b). One Digest (Charlot, 2026) reports bit-identical FP32 transformer forward kernels on Apple Metal, an NVIDIA GPU through Vulkan, WebGPU and three CPU targets, in a self-published technical report; it does not cover training, AMD GPUs or classical learning. HEAL (Zhu et al., 2026) evaluates NVIDIA and AMD GPUs and reduces numerical divergence and answer flips without claiming bitwise identity. None of these shows independent training runs on Apple, NVIDIA and AMD producing the same checkpoint bytes.

Portable GPU programming. Kokkos and Triton provide performance-portable programming models (Edwards et al., 2014; Tillet et al., 2019); Mojo and MAX supply the substrate used here (Lattner et al., 2021; Modular, 2026a; Johnson et al., 2024), with performance portability shown on NVIDIA H100 and AMD MI300A (Godoy et al., 2025). Meganeura (Malyshau, 2026) trains through Vulkan and Metal on NVIDIA, AMD, Apple and Intel GPUs but validates against error thresholds rather than bitwise identity. Mojmelo (yetalit, 2024), a classical-ML library in Mojo for CPUs, has no GPU backend or identity contract, so the shared language alone does not supply the property. The production boosting libraries (Chen & Guestrin, 2016; Ke et al., 2017; Prokhorenkova et al., 2018), GPU tree-boosting research (Wen et al., 2020; 2019; Zhang et al., 2017) and RAPIDS (Raschka et al., 2020) ship GPU learners, but their GPU backends do not support Apple silicon (XGBoost Developers, 2026; LightGBM Developers, 2026; CatBoost Developers, 2026b; NVIDIA, 2026). MacBoost (Rowley, 2026) does train gradient-boosted trees on Apple-silicon GPUs, through Metal kernels written for that one platform. It targets speed on a single vendor and makes no identity claim for training.

10 CONCLUSION

The same machine-learning workload can produce different bits on different GPUs. Mojolearn removes that variable. In its default identical mode, one numerical contract, implemented once in Mojo, gives the same model and output bits across verified Apple, NVIDIA and AMD GPUs and x86-64 and Arm CPUs, for 57 machine-learning algorithms in 12 families, from decision trees to neural networks, in both inference and training. A model trained elsewhere returns the same answers on every verified device, a training run can move between GPU vendors, or be shared among them, without leaving its trajectory, and anyone can check these claims on their own hardware, including against their own CPU. Identity has implementation and execution costs.

Future work. The next steps are making identical mode faster, extending fast mode to every algorithm family, extending coverage to the remaining algorithms and options, verifying more GPUs, and stabilizing the library for a 1.0 release. One route to lower cost is integer arithmetic. Bounded integer accumulation permits different reduction orders while preserving exact results, provided implementations use the same operands, sufficient accumulator width to avoid overflow or saturation, and the same conversion rules. Hardware paths and tilings that preserve those conditions are candidates for further optimization.

REFERENCES

- Al Awar, N. and Singanaboina, S. Y. Controlling floating-point determinism in NVIDIA CCCL. NVIDIA Technical Blog, March 2026. URL <https://developer.nvidia.com/blog/controlling-floating-point-determinism-in-nvidia-cccl>.
- Alves, D. R., Patankar, V., Pereira, M., Stephens, J., Vaziri, N., and Kannan, S. EigenAI: Deterministic inference, verifiable results, 2026. URL <https://arxiv.org/abs/2602.00182>. arXiv:2602.00182 [cs.CR].
- Arun, A., St. Arnaud, A., Titov, A., Wilcox, B., Kolobaric, V., Brinkmann, M., Ersoy, O., Fielding, B., and Bonneau, J. Verde: Verification via refereed delegation for machine learning programs, 2025. URL <https://arxiv.org/abs/2502.19405>. arXiv:2502.19405 [cs.LG].
- Badash, E., Boneh, D., Komargodski, I., and Srivastava, M. Hawkeye: Reproducing GPU-level non-determinism. In *MLSys*, 2026. URL https://proceedings.mlsys.org/paper_files/paper/2026/hash/e217c271a57c365a246b0ad39e668ba8-Abstract-Conference.html.
- Breiman, L. Random forests. *Machine Learning*, 45(1): 5–32, 2001. doi: 10.1023/A:1010933404324.
- Cankaya, N. Bit-exact AI inference verification without performance tradeoffs, 2026. URL <https://arxiv.org/abs/2606.00279>. arXiv:2606.00279 [cs.CR].
- CatBoost Developers. CatBoost gpu histogram and partition-statistics kernels, 2026a. URL <https://github.com/catboost/catboost/tree/7055d33dd10bc5960bc5a32e4d884dab1a1356e8/catboost/cuda>. Source code, revision 7055d33.
- CatBoost Developers. pip install. CatBoost documentation, Python package installation, 2026b. URL <https://catboost.ai/docs/en/installation/python-installation-method-pip-install>. Supported-platforms table lists no GPU support on macOS; accessed September 20, 2026.
- Charlot, D. J. One digest: Bit-identical neural computation across heterogeneous compute fabrics. Technical Report TR-2026-23, Institute for Physical AI @ JBI, July 2026. URL <https://physicalai-bmi.org/assets/papers/one-digest>. Self-published technical report, revised July 24, 2026.
- Chen, B., Wen, M., Shi, Y., Lin, D., Rajbahadur, G. K., and Jiang, Z. M. Towards training reproducible deep learning models. In *ICSE*, pp. 2202–2214, 2022. URL <https://arxiv.org/abs/2202.02326>.
- Chen, T. and Guestrin, C. XGBoost: A scalable tree boosting system. In *KDD*, pp. 785–794, 2016. doi: 10.1145/2939672.2939785.
- Chen, T.-R. The integer alibi: Localizing cross-kernel divergence in INT8-quantized LLM inference, 2026a. URL <https://arxiv.org/abs/2608.13756>. arXiv:2608.13756 [cs.LG].
- Chen, T.-R. Deterministic LLM inference across GPU kernels: Power-of-two INT8 quantization scales and the limits of tolerance-based conformance, 2026b. URL <https://arxiv.org/abs/2609.00363>. arXiv:2609.00363 [cs.LG].
- Collange, C., Defour, D., Graillat, S., and Iakymchuk, R. Numerical reproducibility for the parallel reduction on multi- and many-core architectures. *Parallel Computing*, 49:83–97, 2015. doi: 10.1016/j.parco.2015.09.001.
- Dawson, B. Floating-point determinism, 2013. URL <https://randomascii.wordpress.com/2013/07/16/floating-point-determinism/>.
- Demmel, J. and Nguyen, H. D. Fast reproducible floating-point summation. In *2013 IEEE 21st Symposium on Computer Arithmetic (ARITH)*, pp. 163–172, 2013. doi: 10.1109/ARITH.2013.9.
- Demmel, J., Ahrens, W., and Nguyen, H. D. Efficient reproducible floating point summation and BLAS. Technical Report UCB/EECS-2016-121, EECS Department, University of California, Berkeley, June 2016. URL <https://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-121.html>.
- Donaghy, J., Wilcox, B., Ersoy, O., Rastogi, S., St Arnaud, A., Titov, A., Greenberg, J., Fielding, B., and Grieve, H. OPEN-1B: A fully auditable training run, 2026. URL <https://arxiv.org/abs/2609.17380>. arXiv:2609.17380 [cs.LG], submitted September 15, 2026.
- Edwards, H. C., Trott, C. R., and Sunderland, D. Kokkos: Enabling manycore performance portability through

- polymorphic memory access patterns. *Journal of Parallel and Distributed Computing*, 74(12):3202–3216, 2014. doi: 10.1016/j.jpdc.2014.07.003.
- Gensyn. RepOps, 2025. URL <https://github.com/gensyn-ai/repops-demo>. GitHub repository: bitwise reproducibility of ML operations across hardware targets.
- Gensyn. Introducing REE: Reproducible execution environment, March 2026. URL <https://www.gensyn.ai/news/ree>.
- Geurts, P., Ernst, D., and Wehenkel, L. Extremely randomized trees. *Machine Learning*, 63(1):3–42, 2006. doi: 10.1007/s10994-006-6226-1.
- Godoy, W. F., Melnichenko, T., Valero-Lara, P., Elwasif, W., Fackler, P., Ferreira Da Silva, R., Teranishi, K., and Vetter, J. S. Mojo: MLIR-based performance-portable HPC science kernels on GPUs for the Python ecosystem, 2025. URL <https://arxiv.org/abs/2509.21039>. arXiv:2509.21039 [cs.DC].
- Gond, R., Kamath, A. K., Ramjee, R., and Panwar, A. LLM-42: Enabling determinism in LLM inference with verified speculation, 2026. URL <https://arxiv.org/abs/2601.17768>. arXiv:2601.17768 [cs.LG].
- He, H. and Thinking Machines Lab. Defeating nondeterminism in LLM inference, 2025. URL <https://thinkingmachines.ai/blog/defeating-nondeterminism-in-llm-inference/>.
- Hendel, A. mojolearn: Bitwise-identical GPU machine learning in Mojo on Apple Metal, NVIDIA CUDA and AMD HIP. Zenodo, 2026a. URL <https://doi.org/10.5281/zenodo.22864165>. Fixed evidence archive, evidence-0.8.9-cpu-gpu-20260920-r2.
- Hendel, A. mojolearn: Bitwise-identical GPU machine learning in Mojo on Apple Metal, NVIDIA CUDA and AMD HIP. Zenodo, 2026b. URL <https://doi.org/10.5281/zenodo.22675019>. Version 0.7.0 (alpha-api-0.7.0-20260909), software release archived September 9, 2026.
- Hendel, A. mojolearn: Bitwise-identical GPU machine learning in Mojo on Apple Metal, NVIDIA CUDA and AMD HIP. Zenodo, 2026c. URL <https://doi.org/10.5281/zenodo.22864816>. Version 0.8.10 (alpha-api-0.8.10-20260920), software release archived September 20, 2026.
- Heumos, L., Ehmele, P., Kuhn Cuellar, L., Menden, K., Miller, E., Lemke, S., Gabernet, G., and Nahnsen, S. mlf-core: A framework for deterministic machine learning. *Bioinformatics*, 39(4):btad164, 2023. doi: 10.1093/bioinformatics/btad164.
- Intel. Reproducibility conditions, 2024. URL <https://www.intel.com/content/www/us/en/docs/onemkl/developer-guide-windows/2024-2/reproducibility-conditions.html>. Developer Guide for Intel oneAPI Math Kernel Library, version 2024.2.
- Johnson, E., Lattner, C., Davis, T., Hoge, C., and Clayton, J. MAX is here! what does that mean for Mojo? Modular, February 2024. URL <https://www.modular.com/blog/max-is-here-what-does-that-mean-for-mojo>.
- Ke, G., Meng, Q., Finley, T., Wang, T., Chen, W., Ma, W., Ye, Q., and Liu, T.-Y. LightGBM: A highly efficient gradient boosting decision tree. In *NeurIPS*, 2017.
- Lattner, C., Amini, M., Bondhugula, U., Cohen, A., Davis, A., Pienaar, J., Riddle, R., Shpeisman, T., Vasilache, N., and Zinenko, O. MLIR: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pp. 2–14, 2021. doi: 10.1109/CGO51591.2021.9370308. URL <https://mlir.llvm.org/pubs/>.
- Li, R., Liu, L., Yang, G., Zhang, C., and Wang, B. Bitwise identical compiling setup: prospective for reproducibility and reliability of earth system modeling. *Geoscientific Model Development*, 9:731–748, 2016. doi: 10.5194/gmd-9-731-2016. URL <https://gmd.copernicus.org/articles/9/731/2016/>.
- LightGBM Developers. LightGBM installation guide, 2026. URL <https://lightgbm.readthedocs.io/en/stable/Installation-Guide.html>.
- Lubin, M. and Mueller-Albrecht, R. Floating point conditional numerical reproducibility on CPU and GPU with Intel oneAPI math kernel library (oneMKL). Intel developer article (archived), 2024. URL <https://www.intel.com/content/www/us/en/developer/archive/training/conditional-numerical-reproducibility-cnr.html>. Updated March 25, 2024.
- Malyshau, D. Meganeura: Portable GPU training and inference through Vulkan and Metal, 2026. URL <https://arxiv.org/abs/2608.01563>. arXiv:2608.01563 [cs.LG].
- Modular. Mojo manual, 2026a. URL <https://mojolang.org/docs/manual/>.
- Modular. Mojo system requirements, 2026b. URL <https://mojolang.org/docs/requirements/>. Accessed September 8, 2026.

- NVIDIA. RAPIDS installation guide. NVIDIA RAPIDS documentation, 2026. URL <https://docs.nvidia.com/datascience/install/>. Requires an NVIDIA GPU with compute capability 7.0 or higher on Linux or WSL2; accessed September 20, 2026.
- OpenMD. Reproducibility in parallel OpenMD simulations, February 2013. URL <https://openmd.org/reproducibility-in-parallel-openmd-simulations/>.
- Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., VanderPlas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, É. Scikit-learn: Machine learning in Python. *JMLR*, 12: 2825–2830, 2011.
- Prokhorenkova, L., Gusev, G., Vorobev, A., Dorogush, A. V., and Gulin, A. CatBoost: Unbiased boosting with categorical features. In *NeurIPS*, 2018.
- PyTorch. torch.use_deterministic_algorithms, 2026. URL https://pytorch.org/docs/stable/generated/torch.use_deterministic_algorithms.html. PyTorch stable documentation; accessed September 20, 2026.
- RAPIDS Development Team. cuML: GPU machine learning algorithms, 2026. URL <https://github.com/NVIDIA/cuml/tree/265b9da6a0e75dbef071a3168398b993a5ff6f0e>. version 26.08 source and API documentation.
- Raschka, S., Patterson, J., and Nolet, C. Machine learning in Python: Main developments and technology trends in data science, machine learning, and artificial intelligence. *Information*, 11(4):193, 2020. doi: 10.3390/info11040193.
- Rowley, A. MacBoost: Gradient boosted trees on Apple-silicon GPUs, 2026. URL <https://github.com/alex-rowley/macboost>. Software, version 1.2.0.
- Shi, Y., Ke, G., Chen, Z., Zheng, S., and Liu, T.-Y. Quantized training of gradient boosting decision trees. In *NeurIPS*, 2022.
- Srivastava, M., Arora, S., and Boneh, D. Optimistic verifiable training by controlling hardware nondeterminism. In *NeurIPS*, 2024. URL <https://arxiv.org/abs/2403.09603>. arXiv:2403.09603 [cs.CR].
- TensorFlow. tf.config.experimental.enable_op_determinism, 2024. URL https://www.tensorflow.org/api_docs/python/tf/config/experimental/enable_op_determinism. Page dated April 2024; accessed September 20, 2026.
- Tillet, P., Kung, H., and Cox, D. Triton: An intermediate language and compiler for tiled neural network computations. In *MAPL*, pp. 10–19, 2019. doi: 10.1145/3315508.3329973.
- vLLM Project. Batch invariance. vLLM documentation, 2026. URL https://docs.vllm.ai/en/latest/features/batch_invariance/.
- Wen, Z., Shi, J., He, B., Chen, J., Ramamohanarao, K., and Li, Q. Exploiting GPUs for efficient gradient boosting decision tree training. *IEEE Transactions on Parallel and Distributed Systems*, 30(12):2706–2717, 2019. doi: 10.1109/TPDS.2019.2920131.
- Wen, Z., Liu, H., Shi, J., Li, Q., He, B., and Chen, J. ThunderGBM: Fast GBDTs and random forests on GPUs. *JMLR*, 21(108):1–5, 2020.
- Whitehead, N. and Fit-Florea, A. Floating point and IEEE 754 compliance for NVIDIA GPUs. NVIDIA white paper, 2011. URL <https://docs.nvidia.com/cuda/floating-point/>. Continually updated documentation; accessed September 20, 2026.
- XGBoost Developers. XGBoost GPU support, 2026. URL <https://xgboost.readthedocs.io/en/stable/gpu/>.
- Xie, P., Zhang, X., and Chen, S. RepDL: Bit-level reproducible deep learning training and inference, 2025. URL <https://arxiv.org/abs/2510.09180>. arXiv:2510.09180 [cs.LG].
- yetalit. Mojmelo: Machine learning algorithms in pure Mojo. GitHub repository, 2024. URL <https://github.com/yetalit/Mojmelo>. accessed 2026-08-28.
- Zhang, H., Si, S., and Hsieh, C.-J. GPU-acceleration for large-scale tree boosting. *arXiv:1706.08359*, 2017.
- Zhang, Z., Ding, X., Yuan, J., Liu, R., Mao, H., Xing, J., and Liu, Z. Deterministic inference across tensor parallel sizes that eliminates training-inference mismatch, 2025. URL <https://arxiv.org/abs/2511.17826v2>. arXiv:2511.17826, first posted November 21, 2025; version 2 revised May 29, 2026 [cs.LG].
- Zhu, Z., Thai, L., Yu, S., Liu, Y., Qiao, Y., Wang, C., Xu, H., and Shu, J. Demystifying numerical instability in LLM inference: Achieving reproducible inference for mission-critical tasks with HEAL, 2026. URL <https://arxiv.org/abs/2606.21023>. arXiv:2606.21023 [cs.LG].

A THE VERIFICATION RECORD

The release 0.8.10 record combines separate CPU, Apple Metal, NVIDIA CUDA and AMD HIP runs on fixed fixtures, and each underlying record retains its actual device, source and binary provenance. Of the 7,470 parts in the NVIDIA column, 5,427 were recorded on an H100, 1,611 on an RTX 4090 and 432 on an A100. In the AMD column, 5,616 were recorded on an MI325X and 1,854 on an MI300X. The Apple column is one M4. Of the 7,452 CPU parts, 6,103 were recorded on x86-64 servers, nearly all of them AMD EPYC processors of nine models, and 1,349 on the M4’s Arm cores. The record therefore spans more GPU models than the three named in the table headings: 972 parts also hold the same bits on a second NVIDIA model, and 616 on both AMD models. The reference table contains 273 configurations and 2,457 cells; the complete single-device subset is 214 configurations and 1,926 cells. Every one of its 7,452 CPU numerical parts has the same hash in all three GPU columns. The 18 additional GPU model-writer comparisons also agree. Non-applicability markers count as neither matches nor mismatches.

The archived table decodes each device’s actual value, including explicit values that differ from a table entry’s default reference. The audit checks all applicable parts, not just training hashes. The final supplemental Apple, AMD and NVIDIA collections cover 342, 603 and 648 cells, respectively, with 1,647, 2,187 and 2,934 exact CPU-reference comparisons. These overlap the completed record and are not added to its totals. The supplemental collections fit each cell once; cross-device agreement does not require simultaneous runs, and a single fit does not establish within-device repeated-run stability.

The record is a composite, not one run of one binary. Its columns draw on records from 9 (AMD), 14 (NVIDIA), 21 (Apple) and 28 (CPU) source revisions committed between September 14 and 20, 2026. A record is admitted for a configuration only while that configuration’s numerical revision is unchanged, and each configuration carries that revision in the table. The archive also keeps the superseded records: 6,929 of their hash entries differ from the admitted values, and every one comes from a source revision older than the record admitted for that part. The completed reference table was archived after wheel publication. The archive also retains the exact-wheel publication and installed-check receipts; those packaging checks are distinct from the numerical record. Appendix C identifies the fixed evidence archive.

A.1 The record by part

Table 6 counts the record by what was compared. The first three parts apply where the interface trains, predicts or serializes a model. The others apply where the interface has the property to test, and the verifier records explicit

non-applicability where it does not.

A.2 The record by fixture

Each of the 214 single-device configurations is evaluated on every fixture of Table 8. The table counts numerical agreement across devices separately from whether a source record also repeated a fit.

fixture	what it attacks	parts	divergent
base	the control; ordinary dense input	830	0
ties	integer-grid data, so every distance and split value is tied many ways	830	0
hashed	values derived from a hash, with no structure, ties or symmetry	830	0
wide	column magnitudes from 10^{-4} to 10^4 , where accumulation order matters	830	0
denormal	part of the data below the smallest normal FP32 value	830	0
denormal, flushed	the same data with subnormals flushed to zero, the diagnostic twin	830	0
duplicates	duplicated rows, a constant column and an all-zero column	830	0
odd	12,345 rows and 17 features, which no block, warp or tile size divides	830	0
negative	every value negative, shifted off the origin	830	0

Table 8. The nine fixtures in the single-device record. Each fixture is run by all 214 configurations and contributes 830 matching numerical parts, with zero mismatches. Fixture inputs are identified by hash in the record, and a run whose input hash differs is rejected before any result is compared.

A.3 Negative controls

A negative control is a build of the library with one deliberate arithmetic change, run through the same verifier. It counts for a configuration only when a committed pair of runs, one clean and one changed, on the same device class, differ in at least one part of that configuration’s cells. Table 9 separates observed build changes, harness-only perturbations, declared controls and configurations without a control. The first category establishes sensitivity to changed implementation arithmetic; harness perturbations establish sensitivity of the comparison.

Bitwise-Identical Machine Learning Across Metal, CUDA, and HIP

part	what is compared	configs	parts	parts with equal bits	
				three GPU vendors	and CPU
training	fitted state or the operation’s numerical result	214	1,926	1,926	1,926
saved model	serialized model bytes	148	1,332	1,332	1,314
inference	held-out predictions	183	1,647	1,647	1,647
batch invariance	outputs alone and under supported batch splits	182	1,638	1,638	1,638
serving-scale batch	1,024-row batches and sequence prefixes around chunk boundaries	29	261	261	261
ragged batches	padded sequences equal separate evaluation	20	180	180	180
decode equals forward	cached stepwise decoding equals full forward evaluation	18	162	162	162
gradient accumulation	per-row gradients and reduction-tree-preserving microbatch splits	16	144	144	144
sampler and trainer	sampled and recomputed log-probabilities	20	180	180	180
total			7,470	7,470	7,452

Table 6. Single-device record by numerical part. One part is one compared quantity of one cell, so a configuration contributes nine parts per row, one per fixture; N/A declarations are excluded. The 18 saved-model parts without a CPU value are the categorical and tensor CTR model tables on nine fixtures, where the CPU loads the GPU-written file and does not write a second model.

tier	promise	what it pins
fast	none; speed only. The same fit on the same box may return different bits on two runs	nothing. Float atomics, hardware warp widths, vendor libraries, TF32 matrix product on NVIDIA
deterministic	same box, same build, same input gives the same bits, every run. Says nothing about a second box	every order decided at run time: float atomics, mutex merges, races. Keeps vendor arithmetic and the vendor matrix product
identical (default)	all of the above, and the same bits across Metal, CUDA and HIP	the above, plus denormal policy, FMA contraction, transcendentals, square root, and every geometry taken from a machine constant; closed vendor libraries never called

Table 7. The three numeric tiers. Each rung keeps the rung below it; one source, one flag, no fork.

negative-control evidence	single-device	two-GPU
changed build observed to change numerical bits	205	25
harness perturbation only	0	3
build control declared, no observed numerical change	9	0
no declared or observed control	0	31
total configurations	214	59

Table 9. Negative-control coverage in the archived inventory. Every single-device configuration has a declared control, and 205 of the 214 have been observed to move. A changed build and a harness perturbation establish different properties. An observed change on one fixture or part does not imply that every fixture or part is sensitive to that control.

Three controls illustrate what the comparison can see. Reversing the fold order of the language model’s output-head product changes 9 of 33 recorded losses on every CPU tested. Transposing one operand of the backward matrix product changes 8 of 10 recorded arrays and leaves all two losses equal, which is what corrupting the backward pass

alone should produce, and the comparison names the first wrong tensor with both bit patterns. Zeroing the optimizer moments at a transferred checkpoint changes the following parameters and both moments while preserving the loss, gradients and token inputs. The property parts have controls of their own, which perturb one row of a whole-batch evaluation and must turn every batch-invariance cell from identical to divergent.

A.4 Two-GPU coverage and uncovered options

All 214 single-device configurations have complete numerical coverage across CPU and all three GPU columns. The 18 CPU saved-model writer exceptions are explicit roles, not missing predictions or failed training comparisons. The 59 two-GPU configurations are also complete across the three GPU vendors (Table 10): every numerical part has a value in the Apple, NVIDIA and AMD columns, and no part disagrees. These values were completed after the 0.8.10 release archive was frozen; they are the reference table at repository revision 47372db76, SHA-256 e95fbbb8...d0b7c9. The re-

lease archive itself holds 1,494 of the 1,935 parts complete across the three vendors.

two-GPU numerical-part coverage	parts
CPU and all three GPU columns agree	771
all three GPU columns agree; no numerical CPU role	1,164
incomplete GPU agreement	0
total numerical parts	1,935

Table 10. Coverage of the 59 two-GPU configurations on nine fixtures. Each vendor’s column runs the partitioned plan on one device. These parts are excluded from the single-device totals, and non-applicable roles are not counted.

The source inventory separately enumerates 255 public API entries, including aliases, wrappers and helpers; every entry maps to a verifier configuration. That inventory is not a count of distinct algorithms, and mapping an entry does not establish complete coverage of its options. An interface can accept an option value that no recorded configuration exercises; such a call runs under the same contract but is not a verified result until a configuration records it.

What is counted as an algorithm. Table 1 lists the 57 algorithms by family. An algorithm is a method a practitioner would name. A classifier and a regressor of the same method count once, as do its losses, kernels, distance metrics, penalties, starting rules and sharded two-GPU execution, all of which are configurations. The reference table contains 273 configurations: 214 single-device configurations and 59 two-GPU configurations. Table 10 reports the latter’s coverage. The 24 evaluation metrics, such as accuracy, ROC AUC, silhouette and trustworthiness, are verified the same way and are not counted as algorithms.

A.5 Neural configurations and scale demonstrations

neural group	configs	cells	parts	divergent
transformer, full/windowed/decode	3	27	171	0
Mamba-1	2	18	99	0
Mamba-2	2	18	144	0
Mamba-3	1	9	72	0
hybrid stack	2	18	162	0
byte-level language model	5	45	252	0
MLP	1	9	36	0
BF16 and INT8 weight storage	12	108	648	0

Table 11. Selected neural configurations in the single-device record. Every listed part has equal bits across the Apple, NVIDIA, AMD and CPU columns on all nine fixtures. Reduced-precision weight configurations appear only in the last row. Property coverage is counted separately in Table 6.

Verifier fixtures are small so that anyone can run them. The

byte-level training demonstration below provides a separate long-run check.

A byte-level language model, step by step. A model with two decoder blocks and 34,944 parameters trains for 128 steps on pinned real text on all three vendors, and every recorded array of every step matches: parameters, gradients, AdamW moments, losses, held-out outputs and checkpoint bytes. The step-64 checkpoint resumes on the other vendor in both directions and rejoins the uninterrupted run. On seven CPUs with no GPU present, each given one step’s recorded state, all 640 recomputed gradients, losses, parameters and moments equal the GPU record.

One model trained on three vendors at once. The fixed fold that makes two-GPU training match one GPU also lets GPUs of different vendors, in different machines, train one model together. In the experimental `mojolearn.cross_vendor` interface, which is not part of release 0.8.10, each worker computes the gradients of the logical microbatches it owns, a coordinator sums all of them in microbatch order with the fold the GPU reduction uses, and every worker applies that sum with the same AdamW update. An Apple M4, an NVIDIA H100 and an AMD MI300X, each in its own machine, trained one decoder with 2 blocks, width 32, a 512-token vocabulary and 51,328 parameters for 6 steps, with four microbatches per step (two on the M4 and one on each of the other GPUs). After every step the three workers reported the same SHA-256 of the complete training state (parameters, both AdamW moments and flags), and each state equals the one a single GPU reaches by processing all four microbatches itself.

Runs of the same model in one process reach the same six states on one M4, one H100, two H100 GPUs and one MI300X. A checkpoint written after step 3 by the two H100 GPUs, and one written by the MI300X, each resume on the M4 and rejoin that trajectory. Folding the recorded microbatch gradients under every assignment of microbatches to the Apple, NVIDIA and AMD runs, 1,512 assignments over six steps, reproduces the device sum in every case. The coordinator stops the run when any two workers’ state hashes differ, and it refused a worker that changed a single element of the summed gradient by about 10^{-7} . Every step exchanges full gradients, so the result concerns identity, not throughput.

B THE IDENTICAL-MODE CONTRACT

B.1 Modes and vendor arithmetic

The fixture record measures identical mode. Deterministic mode promises repeatability on one device, while identical mode additionally fixes cross-vendor arithmetic. The unpinned k-NN example in Figure 1 illustrates how vendor square-root behavior can alter bits despite matching

preceding operations. This record supplies no library-wide deterministic-mode mismatch rate.

B.2 Numerical mechanisms

Two boundaries matter when interpreting the record. First, floating-point contraction must be controlled per expression: the Mamba-3 state-rotation fixture requires explicit intermediate rounding barriers, not simply a global policy enabling FMA. Second, host arithmetic can choose device geometry. Host-side arithmetic that selects geometry must therefore obey the same numerical profile; device arithmetic tests alone do not establish this. The portable-arithmetic probes check elementary functions, contraction and subnormal handling at their stated inputs. They do not prove all possible inputs or a future compiler’s behavior.

ledger group	action	entries
order-dependent accumulation	REPLACE	5
a machine query reaching a float reduction	PIN	8
vendor-divergent elementary functions	REPLACE	9
code generation	PIN	1
the IEEE underspecification	PIN	2
ties resolved by arrival order	PIN	4
closed vendor libraries	REPLACE or REFUSE	5
host-side arithmetic that selects a device configuration	PIN	1
iteration counts and caps	PIN	2
random draws	PIN	1
a race, not an order	REPLACE	1
vendor-dependent output order	REPLACE or REFUSE	1
fixed-order floating-point folds	REPLACE	4
sign and extrema conventions	PIN	2
composed algorithms and interface checks	PIN / REPLACE / REFUSE	16
Total		62

Table 12. All 62 numerical-pathway entries in the ledger, each assigned to one group. An entry may describe an operation, a composed algorithm or an interface check; it is not necessarily one code location.

C ARTIFACT

Mojolearn is open source under Apache-2.0 at <https://github.com/mojolearn/mojolearn>. Release 0.8.10 is published on PyPI for macOS arm64 and Linux x86-64 and archived as (Hendel, 2026c), DOI 10.5281/zenodo.22864816; its reference table carries the single-device record unchanged. The numerical record is preserved in the fixed evidence archive (Hendel, 2026a), DOI 10.5281/zenodo.22864165. It includes the canonical table, the machine-readable audit, the

raw-record manifest and all 548 referenced record entries, plus release receipts. The paper audit verifies the manifest hashes and the 29,862 recorded numerical column values underlying the single-device result. The project archive uses concept DOI 10.5281/zenodo.22068632.

Implementation size. Table 13 counts the source at revision 88e0305f with cloc 2.06, code lines only, over every Git-tracked file in a programming language; data and prose files are excluded. The Mojo is divided by role. The library row holds the estimators, kernels and bindings. The verification row holds the host reference implementations the GPU kernels are compared against, the per-family checks, the fixtures and the sabotage arms that must fail before a result counts; 684 source files outside the result directories name a sabotage arm. Verification is 1.4 times the size of the library it checks. Python holds the public package, its tests and the gate, tooling and packaging scripts. The result directories retain another 2,292 lines of copied scripts beside the evidence they produced, and the table excludes them. Code size describes implementation scope, not numerical coverage or correctness.

language and role	files	code lines
Mojo library (estimators, kernels, bindings)	823	176,999
Mojo verification (oracles, checks, fixtures, sabotage arms)	614	239,659
Mojo benchmarks	42	13,084
Python package	112	37,378
Python tests	157	23,983
Python gates, tooling and packaging	426	75,271
Python benchmarks	27	4,946
Python reference scripts	54	6,194
shell	371	39,455
C++ and headers	23	3,228
YAML and TOML configuration	18	2,716
total	2,667	622,913

Table 13. Source by language and role at revision 88e0305f. Code lines exclude comments and blanks. The Mojo rows sum to 429,742 and the Python rows to 147,772.