DIFFDOG: FORMALLY VERIFIED AUTOMATIC DIFFERENTIATION AT BILLION-PARAMETER SCALE

Anonymous authors
Paper under double-blind review

ABSTRACT

As AI training runs grow in scale, cost, and complexity, software errors become increasingly consequential. Yet mainstream automatic differentiation (AD) libraries at the heart of these training runs rely on Python-centered stacks that are unable to provide machine-checkable correctness guarantees. We present DIFFDOG, a formally verified AD system in Lean that supports arbitrarily nested forward- and reverse-mode AD, automatic vectorization, and compilation to GPU and TPU executables, with machine-checked correctness proofs for its tracing, differentiation, and vectorization transformations. Programs written against DIFFDOG's abstract array interface can be instantiated over mathematical value types for direct reasoning and over traceable array types for compilation to accelerators. This dual interpretation provides an extensible foundation for developing verified models, numerical algorithms, and program transformations without relinquishing high-performance accelerated execution. We exhibit DIFFDOG's expressiveness and utility by training a language model with verified gradients, introducing the first accelerator-executable SGD optimizer with verified $O(1/\sqrt{T})$ convergence, and demonstrating that DIFFDOG automatically synthesizes a counterexample to Adam's original convergence claim, independently exposing the failure identified by Reddi et al. (2018). Finally, DIFFDOG's training and inference throughput is competitive with optimized JAX and PyTorch implementations on leading open-weight models, including Qwen 3.8-27B and Gemma 4.

1 INTRODUCTION

Artificial intelligence workloads are growing exponentially. Despite the transformative expansions in compute and physical infrastructure supporting this growth, the numerical computing software stack that underlies these training runs has remained largely unchanged. Numerical computing software is uniquely challenging to implement correctly: whereas errors in conventional software may manifest as crashes, faults in numerical software can silently perturb results (Kanewala & Bieman, 2013; Peng et al., 2021; Hatton & Roberts, 1994; Hook & Kelly, 2009), potentially resulting in errors in AI training runs being discovered post-hoc or not at all. And numerous bugs have been discovered in modern AI training software (Wang et al., 2022; Chen et al., 2023; Humbatova et al., 2020; Pham et al., 2019; Wei et al., 2022; Gu et al., 2022; Liu et al., 2023b;a).

Formal verification offers a fresh avenue forward: connect executable programs with machine-checkable proofs that they satisfy formally-defined specifications. Developing formally verified software is challenging (Ringer et al., 2019) but promises to eliminate entire classes of bugs. In principle, a verified AI software stack could eliminate bugs from training recipes, models, and algorithms. Existing verified AD systems have established the possibility of verification in ML (Selsam et al., 2017; George et al., 2026b). Our goal is to bring these guarantees to demanding real-world workloads.

In this paper we explore the development of a formally verified automatic differentiation system with support for whole-program compilation to run on accelerators. We present DIFFDOG, a formally verified AD system that connects mathematical reasoning and accelerator execution through a shared interface, making verified differentiation useful for realistic ML programs. DIFFDOG features static array shape checking, forward, reverse, mixed-mode, and nested automatic differentiation, a JAX- and NumPy-compatible API, and automatic vectorization à la `vmap`.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

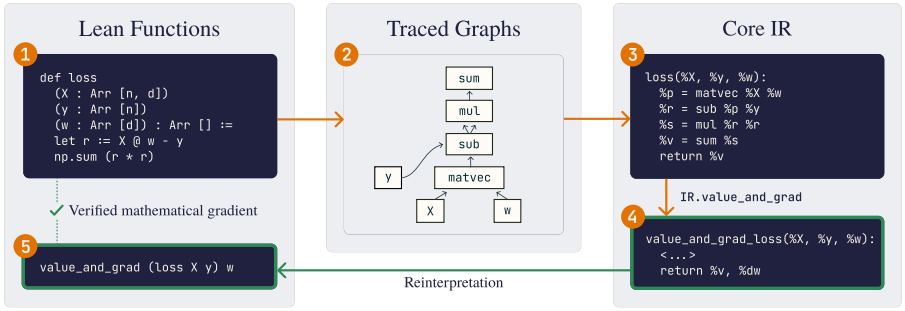


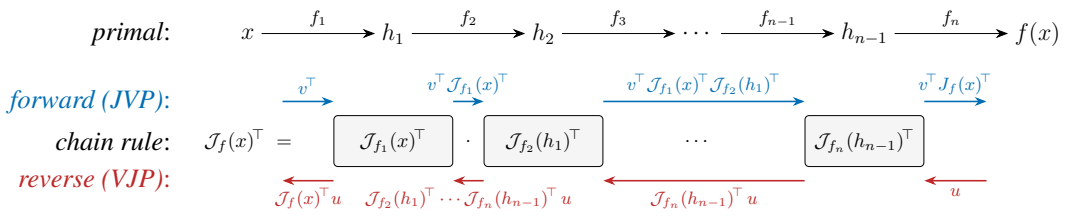
Figure 1: **DIFFDOG offers formally verified automatic differentiation without compromising performance.** DIFFDOG runs billion-parameter open-weight models as fast as JAX and PyTorch. Unlike conventional AD systems, DIFFDOG provides machine-checked proofs of its AD correctness and enables users to verify properties of their own programs, including compile-time checking of array shapes and optimizer convergence rates. DIFFDOG works by transforming programs into a core intermediate representation language, rewriting, and then reinterpreting to produce new programs.

The central design of DIFFDOG is a polymorphic array interface that connects mathematical reasoning with executable programs. Users write ordinary Lean functions against this interface, then instantiate the same program over concrete values for evaluation, mathematical arrays for reasoning, or symbolic arrays for program transformation and compilation. Achieving this design in practice requires efficient tracing that respect's Lean's programming model: shared computations must remain shared, despite Lean's referentially transparent programming model not exposing pointer identity. We develop an expected-linear-time tracing method that preserves sharing and works entirely within Lean's constraints. Tracing enables us to reify user functions into programs in an intermediate representation that is then used for differentiation, auto-vectorization, and compilation through OpenXLA for accelerated execution. As a result, DIFFDOG runs billion-parameter models with performance competitive with JAX and PyTorch. We additionally show that DIFFDOG is capable of pretraining and fine-tuning a 125M-parameter GPT-2-style language model.

We demonstrate the connection between executable programs and verification through several formalizations. Our AD implementation formally satisfies a "cheap gradient" theorem in the style of Baur & Strassen (1983). DIFFDOG automatically synthesizes a counterexample to Adam's faulty convergence argument, independently recovering the error discovered by Reddi et al. (2018). In contrast, we implement gradient descent in DIFFDOG and verify that our implementation enjoys $O(1/T)$ and $O(1/\sqrt{T})$ convergence rates in the full- and mini-batch settings, respectively.

2 BACKGROUND

Automatic differentiation (AD) is the transformation of programs into programs that calculate their mathematical derivatives. Consider as a simplified example a function $f : \mathbb{R}^m \to \mathbb{R}^n$ that is the composition of a sequence of smaller functions $f_1, \ldots, f_n$, $f(x) = (f_n \circ \cdots \circ f_2 \circ f_1)(x)$. Recall that the Jacobian, $\mathcal{J}_f(x) \in \mathbb{R}^{n \times m}$, is the closest linear approximation to the "primal" function f at the point x. By the chain rule, Jacobians multiply and automatic differentiation can be seen as the programmatic calculation of terms through the Jacobian sequence.



Thanks to the associativity of multiplication, we are free to select the association order of this sequence of Jacobian multiplications, with considerable consequences for the computational cost. In full generality, programs are in fact directed acyclic graphs of Jacobians and intermediate values.

Alas, in this case, solving for the optimal association is NP-hard (Naumann, 2008) but nevertheless two reasonable options exist. Forward mode pushes tangents $v^\top$ forward through the Jacobian sequence, $(((v^\top \mathcal{J}_{f_1}^\top)\mathcal{J}_{f_2}^\top)\dots\mathcal{J}_{f_n}^\top)$. Conversely, reverse mode pulls cotangents u back through the computation graph in reverse order, $\mathcal{J}_{f_1}^\top(\mathcal{J}_{f_2}^\top(\dots(\mathcal{J}_{f_n}^\top u)))$. As a general rule, prefer forward mode when $m < n$ and reverse mode otherwise. Combining forward and reverse mode in different subgraphs allows us to recover arbitrary association orders if desired, so-called mixed-mode AD (Griewank & Reese, 1991).

Whereas forward mode follows the same path as the primal computation, reverse mode must walk backwards on terms that depend intermediate values h_i, and therefore must replay the primal computation as a prerequisite. Note a subtle control flow implication: cotangents must be accumulated at each node before the reverse mode walk can proceed beyond a node. Whereas forward mode can directly follow the execution of the primal computation, this wrinkle in reverse mode complicates the implementation and verifyication of reverse mode AD.

We study the implementation of a *formally verified* AD system in particular. Formal verification is the practice of writing machine-checkable proofs of correctness for programs. Whereas conventional programming centers on the expression of executable programs, formal methods commingle and connect executable programs with machine-checkable mathematical specifications and proofs. In particular, we work in the Lean language (de Moura & Ullrich, 2021). As a dependently typed language, Lean enables users to express properties of runtime values in the compile-time type system, breaking the traditional boundary between values and types. For example, we make use of this to encode array shapes in their types, preventing illegal shape errors at runtime.

We study verification and correctness with respect to the real number semantics of input programs. In other words, does the program return the correct result on an idealized computer operating on genuine real numbers? We connect our formalizations with Mathlib's calculus definitions to ground them to a trusted foundation (The mathlib Community, 2020). Inevitably we cannot verify everything; for our purposes we trust and do not attempt to verify the correctness of floating point numerics, the Lean runtime and compiler, machine learning compilers, external libraries and drivers (e.g. CUDA), and hardware.

3 A Tour of DiffDog

User programs in DiffDog are plain Lean functions from arrays to arrays. For example,

```
def loss (...) (X : Arr [n, d])
               (y : Arr [n])
               (w : Arr [d]) : Arr [] :=
  let r := X @ w - y
  np.sum (r * r)
```

where `Arr : Shape → Type` denotes a polymorphic array type, allowing for the program to be instantiated to operate on mathematical arrays for formal reasoning, symbolic tracer types for translation to IR, or any other type satisfying an array capability interface. Note the `np.sum` harkening to NumPy: We strive to maintain API compatibility with existing JAX and NumPy APIs in the interest of minimizing user friction (Bradbury et al., 2018; Harris et al., 2020). DiffDog's implementation differs from JAX and NumPy in that array shapes are dependently typed, eliminating runtime shape mismatch errors. While a NumPy-compatible operation set is provided by default, it does not benefit from any privileged status internally, and users can extend DiffDog with custom operations.

Core DiffDog transformations, like `grad` and `vmap`, accept these functions as input and return new functions by way of tracing the input functions to a core intermediate representation (IR), transforming programs in this IR, and then reinterpreting to produce transformations of the input Lean function.[1] To explore this process in detail, we follow `loss` in its journey through `value_and_grad (loss X y) w`.

[1] We attempted to avoid reifying functions into an IR, and instead implement AD via control flow (Wang et al., 2019) but found this approach to be ill-suited to Lean's programming model.

3.1 Tracing and the Core Intermediate Representation

To manipulate control flow for transformations like reverse-mode AD and program compilation, we must first reify user programs as data (Danvy, 1996; Berger & Schwichtenberg, 1991). We utilize tracing as a means of converting a function into values in the core IR (Frostig et al., 2018). Logically, we treat user programs as directed acyclic computation graphs (DAGs) with nodes representing operations. In contrast to languages like C++ and Python, Lean is referentially transparent, meaning that values in the language are not embellished with any notion of identity, e.g. a heap address. Consequently, conventional DAG data structures and tracing algorithms are not representable in Lean.

For example, both operands of `r * r` refer to the same value, and tracing must preserve this sharing in order to avoid duplicating the computation `X @ w - y`. With residual connections, a failure to respect sharing results in an exponential blowup such that even a post-hoc deduplication phase is infeasible.

Our tracing solution operates in two stages. First, we execute the function on symbolic array values producing a traced computation graph. These symbolic arrays record their provenance and track their input arguments by value, meaning that the resulting output value captures the complete computation graph. The Lean runtime represents this DAG compactly in memory via pointer aliasing, but from the user's perspective they constitute terms of potentially exponential size and are therefore not suitable for direct use. Crucially, we also endow each symbolic array with an identity in the form of a Merkle tree hash.

Next, we recursively walk the computation graph rooted at an input symbolic array, interning values into an ordered hash-cons arena along the way (Filliâtre & Conchon, 2006; Ershov, 1958; Goto, 1974). Although a naïve recursive walk of the symbolic array tree could result in an exponential time blowup, we invoke a trick from Selsam et al. (2020). They introduce `withPtrEq`, now part of Lean's standard library. It is functionally equivalent to Lean's standard equality checking but includes short-circuiting behavior when the Lean runtime identifies that the inputs are pointer-equal. When walking the traced computation graph, we rely on `withPtrEq` to skip already-interned subgraphs in constant time, resulting in an expected linear time traversal for the complete DAG. We emphasize that the pointer equality comparison provided by `withPtrEq` serves as a performance optimization and is not load-bearing with respect to the correctness of the algorithm.[2]

The ordered hash-cons arena is then translated into the core IR, a dependently-typed A-normal form (ANF) representation (Flanagan et al., 1993). Care is taken to preserve topological ordering of the underlying DAG in this lowering. A detailed walk through of the core IR is offered in Appendix A.2. For `loss`, the resulting IR program is shown schematically in Listing 1.

```
loss(%X, %y, %w):
  %p = matvec %X %w
  %r = sub %p %y
  %s = mul %r %r
  %v = sum %s
  return %v
```

Listing 1: Example core IR.

3.2 Transformation and Reinterpretation

With the user program reified into IR terms, we proceed to manipulate it as data. Transformations are defined on a per-`Op` basis. We then combine individual `vjp` rules via induction on instruction sequences.

```
class ADable (Op : Sig) (Host : Sig) where
  vjp : {ins : List Shape} → {out : Shape} → Op ins out
        → ANFFun Host (out :: ins) ins
  jvp : {ins : List Shape} → {out : Shape} → Op ins out
        → ANFFun Host (ins ++ ins) [out]
```

Here, `Op` is the operation under study, and `Host` is the signature "hosting" the derivative graph, generally larger than `Op`, and `ANFFun` is the type of programs in IR. Note that `ADable` is not

[2]For the referential transparency hard-liners we additionally offer a purely referentially-transparent tracing algorithm that does not rely on any pointer equality, but runs in worst case cubic time.

parameterized on the operational array type and is instead defined syntactically to prevent aberrant instances from defining conflicting rules for different array types.

Recursion over instruction sequences with these per-operation differentiation rules gives us reverse mode differentiation of programs in the IR. At each step the appropriate `vjp` rule is applied from the matching `ADable` instance, growing a new IR program. It is in this step that control flow is reversed such that cotangents are fully accumulated before processing each node. In our running example, the reverse rule for `mul %r %r` contributes twice to the cotangent of `%r`, once for each operand. These cotangents are added before continuing the reverse process.

In addition, a canonical interpreter is also defined for programs in the IR, giving a translation from programs as values to programs as Lean functions.

The path for `value_and_grad (loss X y) w` is then clear: first `loss X y` is traced with a symbolic array in place of `w`, resulting in a program in the IR. This IR is then transformed according to differentiation rules provided by `ADable`, and finally the resulting IR is reinterpreted, producing a new Lean function.

3.3 Verified Correctness

To verify AD results, we connect two guarantees: tracing preserves a function's mathematical meaning and differentiation of IR programs computes their derivatives.

Crossing the chasm between functions and IR presents a challenge in Lean. Centering the API on plain functions prevents the formalization of a naturality theorem for tracing, universally quantified. In other words, limitations in Lean make it impossible to construct a theorem stating that for all functions f, the trace-to-reinterpretation of f is equal to the evaluation of f instantiated at concrete values. Instead, we address this by verifying the connection separately for each concrete function with a macro that automatically establishes end-to-end equality across the tracing and reinterpretation boundaries. For example, this macro automatically rejects expression-identity hash collisions at compile time.

Specifying correctness of the IR transformation is more straightforward. We specify correctness with respect to Mathlib's Fréchet derivative, `fderiv` (The mathlib Community, 2020). We introduce type classes `FDerivDomain`, `FDerivLaw`, and `AdjointLaw` to provide executable descriptors of `Op` domains, equality of the `jvp` and Mathlib's Fréchet derivative, and proofs that `jvp` and `vjp` are mutually adjoint, respectively. As with `ADable`, these are defined per-`Op` and composed into a larger theorem that ultimately establishes

```
vjp f x = (fderiv ℝ (denote f) x)†
```

where `†` is Mathlib's adjoint of linear maps and `denote f` is the denotational meaning of `f`, traced and specialized to $\mathbb{R}$'s. We take special care to avoid discontinuities and irregularities, e.g. $\mathrm{ReLU}(0)$, by tracking and requiring all inputs to be in-domain with `RegularAt`.[3]

Similarly, other transformations provide their own correctness statements. For example, our automatic vectorization primitive `vmap` is proven to be identical to evaluating the function on slices of the input and concatenating the results along the batch dimension.

We additionally note that tracing nested AD programs risks perturbation confusion (Siskind & Pearlmutter, 2005; 2008), the accidental combination of incoherent symbolic arrays belonging to different tracing levels. We wrap symbolic values in a rank-2 type, akin to the `runST` trick, to address this concern (Launchbury & Jones, 1994). This resolves perturbation confusion issues, albeit at the cost of relying on metatheory. Concretely, this precludes writing a Lean theorem stating the correctness

[3] We do not assert AD correctness where the program is said to be irregular according to `RegularAt`. An adversarial op author could therefore artifically expand `RegularAt` and hide incorrect behavior in that region. This could be mitigated adding constraints to `RegularAt` but this is not an attack we have focused on.

of nested AD. After experimenting with an alternative graph-based frontend we found the ergonomic benefits of operating on plain functions outweigh the loss of an internal nested AD theorem.

3.4 Compilation

We leverage OpenXLA (OpenXLA Project, 2026) for compilation to accelerators, including GPUs and TPUs. Programs are traced to the core IR, compiled from the core IR to StableHLO, OpenXLA's preferred input MLIR dialect (Lattner et al., 2021), and then compiled by OpenXLA to run on device. We expose these compiled executables to the user as seamless Lean functions accepting and returning ordinary Lean values.

The same value-and-grad result can be compiled to run directly on device, as in

```
do
  let loss_fast ← xlajit (value_and_grad (loss X y))
  let out ← loss_fast w
  IO.println s!"{out}"
```

As a side-effectful operations `xlajit` and its outputted function operate in the `IO` monad.

4 Applications and Evaluation

We evaluate DiffDog's suitability as a platform for machine learning program verification and theorem proving, its capacity to train a GPT-2-style language model from scratch, and its performance on training and inference workloads for Qwen 3.8-27B and Gemma 4 26B A4B models.

4.1 Verification Case Studies

We exhibit three case studies for the use of DiffDog for reasoning about executable programs: proving optimizer convergence, automatically synthesizing a counterexample to an optimizer's convergence argument, and bounding the size of transformed derivative graphs.

(Stochastic) gradient descent with verified convergence rates. We implement the gradient descent update in DiffDog and connect it with verified convergence rates in the full- and stochastic-gradient settings, $O(1/T)$ and $O(1/\sqrt{T})$ respectively. We formalize these convergence rates with respect to established Mathlib definitions of convexity, Lipschitz smoothness, and probability theory.

```
theorem gd_rate (...) :
  f (gradientDescent (EuclideanArray.scalar gamma) gradient x0 T) -
    f xStar ≤ ‖x0 - xStar‖ ^ 2 / (2 * gamma * (T : ℝ))

theorem sgd_rate (...) :
  (𝔼 ω : Draws n T,
    (objective loss (pathMean x0 (gamma0 / Real.sqrt (T : ℝ)) grad ω) -
      objective loss xStar)) ≤
    ‖x0 - xStar‖ ^ 2 / (gamma0 * Real.sqrt (T : ℝ)) +
      (2 * gamma0 * gradientNoise loss grad) / Real.sqrt (T : ℝ)
```

To our knowledge these are the first GPU-executable optimizers with verified convergence rates.

Automatic synthesis of a counterexample to the Adam convergence argument. Verification can also expose mistaken claims about optimizer behavior. Kingma & Ba (2015) introduced the Adam optimizer, touting its superior empirical performance and a convergence analysis claiming $O(1/\sqrt{T})$ average regret in the stochastic convex optimization setting. Since its introduction Adam and its variants have become a popular component of modern machine learning practice. However, Reddi et al. (2018) later found that the purported convergence proof was flawed and in fact Adam fails to converge at all on certain functions, upending the theoretical foundation of one of the field's most popular algorithms.

```
example (α : ℝ) (hα : 0 < α) : ∀ (β : ℚ)
    (g₁ g₂ : LeanArray ℚ [2, 2]),
    0 < β → β < 1 →
    (matrixIx g₁).1.1 ≠ 0 → (matrixIx g₁).1.2 ≠ 0 →
    (matrixIx g₁).2.1 ≠ 0 → (matrixIx g₁).2.2 ≠ 0 →
      arrayGammaDiagonalNonneg α β g₁ g₂ := by
  intro β g₁ g₂ hβ0 hβ h00 h01 h10 h11
  rw [arrayGammaDiagonalNonneg_iff α hα β hβ0 hβ]
  clear α hα
  plausible
```

```
Found a counter-example!
β := (2 : Rat)/3
g₁ := [[1, (1 : Rat)/4], [2, -1]]
g₂ := [[0, 0], [-1, (-1 : Rat)/2]]
...
issue: ¬(0 < (1 : Rat)/48) does not hold
```

Figure 2: DIFFDOG and Plausible automatically find a counterexample to Adam's $\Gamma_t \succeq 0$ claim.

We implement Adam in DIFFDOG, translate its convergence argument, and demonstrate that DIFFDOG and the Plausible tool (The Plausible contributors) automatically synthesize a counterexample to Adam's faulty convergence argument, independently recovering the result from Reddi et al. (2018). The object of interest is the difference between square roots of second-order moments: $\Gamma_t = \sqrt{V_t}/\alpha_t - \sqrt{V_{t-1}}/\alpha_{t-1}$ where V_t is a diagonal matrix of second-order estimates and α_t is the step size. Whereas Adam errantly assumes that $\Gamma_t \succeq 0$, DIFFDOG and Plausible automatically produce a counterexample (Figure 2).

Cheap-gradient bound for an arithmetic signature. Besides user programs like optimizers, DIFFDOG supports reasoning about program transformations themselves. For example, the Baur-Strassen theorem bounds the circuit complexity of scalar AD (Baur & Strassen, 1983). Commonly referred to as the "cheap gradient" principle, the effect is that derivatives can be calculated by circuits that are linear in size with respect to the input circuit.

Theorem 1 (Cheap gradient theorem)**.** *For a scalar arithmetic circuit calculating $f(x_1, \ldots, x_n)$ using operations $\{+, -, \times, /\}$ over $\mathbb{R}$,*

$$S\left(f, \frac{\partial f}{\partial x_1}, \ldots, \frac{\partial f}{\partial x_n}\right) = O(S(f))$$

where S denotes the circuit size.

In DIFFDOG we formalize that our executable AD implementation follows a cheap gradient bound for programs written in an arithmetic `Sig`, showing that the length of the gradient `ANFSeq` is linear in terms of the length of the input program `ANFSeq`. We list the formalized theorem in Appendix A.4.

4.2 TRAINING A LANGUAGE MODEL FROM SCRATCH

We pretrain and fine-tune a 125M parameter GPT-2-style model, following the nanochat recipe (Radford et al., 2019; Karpathy, 2025). We pretrain on a 503M token subset of the ClimbMix dataset (Diao et al., 2025). We apply the Muon optimizer (Jordan et al., 2024) on attention and MLP weights, and the AdamW optimizer (Loshchilov & Hutter, 2019) on token embeddings. The model code, optimizers, and training loop are fully implemented in Lean and DIFFDOG.

Following pretraining, we complete a supervised fine-tuning phase with data collected from the SmolTalk ($1\times$ repeats), GSM8K ($4\times$ repeats), and MMLU datasets ($3\times$ repeats) (Allal et al., 2025; Cobbe et al., 2021; Hendrycks et al., 2021). In total this constitutes 490M tokens of supervised fine-tuning data. These results are presented in Figure 3. Example generations are provided in Appendix A.5.

4.3 PERFORMANCE

We evaluate performance on two axes: core automatic differentiation workloads and GPU-accelerated training and inference with popular open-weight models.

4.3.1 AD PROGRAM CONSTRUCTION

We compare DIFFDOG's reverse mode automatic differentiation performance compared to JAX and PyTorch. For variable depths of the nanochat GPT-2-style model in Section 4.2, each implemen-

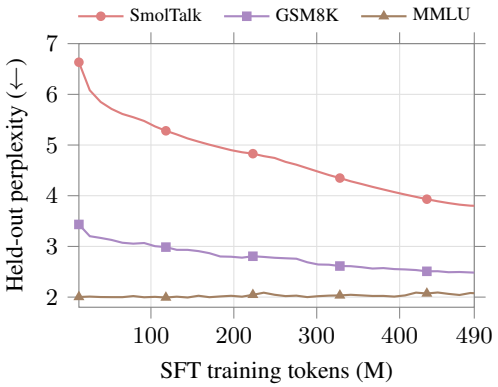


Figure 3: **Fine-tuning a 125M parameter GPT-2-style model in DIFFDOG.** Held-out task perplexities over the course of fine-tuning. We showcase example conversations with the resulting model in Appendix A.5.

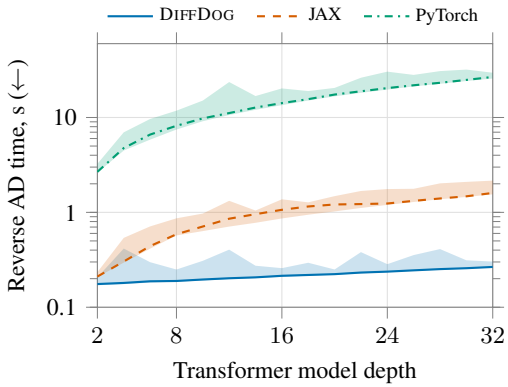


Figure 4: **DIFFDOG constructs derivative programs faster than JAX and PyTorch.** Median reverse mode AD time for a GPT-2 model. At depth 32 DIFFDOG runs in 266ms, 6× faster than JAX and 101× faster than PyTorch.

tation was scored on the time required to produce the backwards pass, specifically an `ANFSeq` in DIFFDOG, a `jaxpr` in JAX, and `ATen FX` in PyTorch. We do not benchmark the compilation of the backwards pass to executables in order to isolate AD performance. See Figure 4.

4.3.2 BILLION-PARAMETER EXECUTION

To demonstrate DIFFDOG's ability to run leading billion-parameter, open-weight models, we evaluate its performance on the dense Qwen 3.8 27B model (Qwen Team, 2026) and the mixture-of-exports Gemma 4 26-A4B model (Abd et al., 2026). We compare against JAX and PyTorch implementations, using their official releases where available and making the adaptations described in Appendix A.8.

We consider three representative workloads: supervised fine-tuning setup with LoRA adapters (Hu et al., 2022), inference prefill, and inference decoding with key-value caching. Performance on these benchmarks depend primarily on the program emitted by the AD implementation and the underlying ML compiler. For DIFFDOG and JAX, StableHLO programs are emitted and compiled by OpenXLA. For PyTorch, the approximate analogy is `ATen FX` programs compiled by TorchInductor and Triton. To minimize confounds, DIFFDOG and JAX share a single byte-identical OpenXLA plugin build, along with the same `ptxas` and cuDNN versions and the same device memory limit, so that comparison turns on the emitted program rather than the compiler or the runtime. On Gemma the two also share a mixture-of-experts kernel, as described below. On every decode run of both models, all three stacks emit identical greedy token sequences, all 128 tokens of them. We attempted to benchmark TorchLean (George et al., 2026b) as well but found that it was missing necessary operations and could not support loading weights at 27B parameter scale.[4]

Table 1 summarizes training, prefill, and decoding throughput results. DIFFDOG generally performs near the best baseline. Qwen decoding remains slower than PyTorch. Notably, DIFFDOG does not have performance identical to JAX despite both compiling through OpenXLA. This is attributed to slight differences in the emitted StableHLO programs and DIFFDOG's lack of in-place updates. JAX offers the option to "donate" arrays in `jax.jit`, allowing OpenXLA to skirt the cost of fresh allocations by performing updates in-place, a feature not yet implemented in DIFFDOG. Example generations from DIFFDOG inference are listed in Appendices A.6 and A.7.

[4] At the time of writing, TorchLean reads weights as JSON arrays of floats. For a 27B parameter model, this ASCII representation would require ∼500GB of disk space. Additionally, TorchLean's process of loading weights onto GPU materializes them in float64 precision which would require 216GB of host RAM.

Table 1: **Performance on billion-parameter open-weight models is competitive with existing state-of-the-art systems.** Throughput presented in tokens per second. Higher is better. Percentages indicate throughput shortfall relative to the best system in each model/workload comparison.

Model	System	LoRA Training	Prefill	Decode
Qwen 3.8 27B	DIFFDOG (ours)	**521 tok/s**	**1103 tok/s**	41 tok/s (-18%)
	JAX	513 tok/s (-2%)	1097 tok/s (-1%)	42 tok/s (-17%)
	PyTorch	453 tok/s (-13%)	938 tok/s (-15%)	**50 tok/s**
Gemma 4 26-A4B	DIFFDOG (ours)	**1356 tok/s**	**3425 tok/s**	139 tok/s (-5%)
	JAX	1343 tok/s (-1%)	2231 tok/s (-35%)	**146 tok/s**
	PyTorch	1181 tok/s (-13%)	2077 tok/s (-39%)	137 tok/s (-6%)

5 RELATED WORK

Formal methods in machine learning and AI. Certigrad (Selsam et al., 2017) was the first to develop a formally verified implementation of automatic differentiation. SciLean (Skřivan, 2024) is a scientific computing library in Lean, including a formally verified source-to-source AD implemented via metaprogramming. de Vilhena & Pottier (2023) implement AD with effect handlers and explore its verification. Hyman et al. (2026) implement a verified, imperative heap-based reverse mode AD algorithm in Isabelle/HOL (Bulwahn et al., 2008). Recently, George et al. (2026b) develop TorchLean, a verified neural network framework supporting AD and neural robustness certificates. In contrast to prior work, we explore the design of an AD system that simultaneously offers formally verification footholds for user programs and supports realistic workloads with large, practically-relevant models. We defer a detailed comparison with existing AD systems to Appendix A.1.

Li et al. (2025) verified mathematical models of popular convex optimizers, including gradient descent. Spasic & Maric (2012) implement a verified simplex solver. CvxLean (Bentkamp et al., 2023) studies a verified reduction engine for convex programming problems.

Automatic differentiation in machine learning. We refer readers to Baydin et al. (2017) for an overview of AD in machine learning. Forward mode AD is attributed to Wengert (1964). Reverse mode AD is attributed variously to Werbos (1974); Linnainmaa (1976); Speelpenning (1980). Rumelhart et al. (1986) popularized reverse mode AD in the machine learning community. Consider Griewank (2012) for a historical rabbit hole.

Machine learning has been blessed with a variety of AD tools (Collobert et al., 2011; Jia et al., 2014; The Theano Development Team, 2016; Abadi et al., 2016; Paszke et al., 2017). At the time of writing, modern practice is dominated by JAX (Bradbury et al., 2018) and PyTorch (Paszke et al., 2019). We highlight Maclaurin et al. (2015) for initiating the function-based lineage of AD design.

6 DISCUSSION AND FUTURE WORK

We explore the design and implementation of a formally verified automatic differentiation system in Lean. Building against a shared array interface, our system, DIFFDOG, simultaneously offers comprehensive AD support of plain user functions, program verification, and competitive performance. Users are thus enabled to mathematically reason on the same program definitions that execute on device.

Still, important work still remains including support for non-float32 data types, multi-host and multi-accelerator support, and development of in-graph pseudorandom number generation.

We hope that this work may serve as a beachhead towards a formally verified numerical programming software ecosystem, and provide foundational tooling for others' work in this space. We are especially excited by the prospect of DIFFDOG improving the efficacy and reliability of autonomous research agents.

AI USE STATEMENT

In this work, we used generative AI tools for implementing methods. We have not used generative AI tools for help developing theoretical models or conceptual frameworks, formulating mathematical claims, providing critical ingredients for proving mathematical claims, assistance in the writing of proofs, proposing or refining hypotheses, designing or providing feedback on research methodology or experiments, assistance with translation, supporting qualitative and thematic data analysis, nor interpreting results. and generating synthetic data sets, and cleaning and reformating datasets are not applicable to this work. Additionally, we used generative AI tools for creating figures 3 and 4, and coding assistance . We have reviewed all AI-assisted work, and take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

We emphasize that the paper was written by-hand. AI feedback was considered only after a complete draft was finalized. Proposals were weighed carefully, and some resulted in human edits. Figure 1 was hand-constructed.

REPRODUCIBILITY STATEMENT

We open source DIFFDOG at `https://github.com/REDACTED/REDACTED`. Experimental details for each of our results can be found in Appendix A.8.

Note to reviewers: We provide a copy of the codebase in the submission materials. DIFFDOG is an actively developed project so in the interest of reproducibility we have also provided the current state of all open pull requests at the time of submission. We view the core DIFFDOG implementation as stable.

REFERENCES

Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pp. 265–283. USENIX Association, 2016. URL `https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi`.

Sherif El Abd, Vaibhav Aggarwal, Robin Algayres, Alek Andreev, Olivier Bachem, Ian Ballantyne, Cormac Brick, Victor Carbune, Michelle Casbon, Mayank Chaturvedi, Aditya Chawla, Victor Cotruta, Alice Coucke, Phil Culliton, Robert Dadashi, Lucas Dixon, Mohamed Elhawaty, Utku Evci, Clément Farabet, Johan Ferret, Filippo Galgani, Sertan Girgin, Jean-Bastien Grill, Maarten Grootendorst, Jiaxian Guo, Cassidy Hardin, Yanzhang He, Steven M. Hernandez, Omri Homburger, Léonard Hussenot, Ju-yeong Ji, Armand Joulin, Aishwarya Kamath, Parnian Kassraie, Olivier Lacombe, Preethi Lahoti, Gaël Liu, Gus Martins, Luciano Martins, Tatiana Matejovicova, Ramona Merhej, Nikola Momchev, Sneha Mondal, Ryan Mullins, Sindhu Raghuram Panyam, Shreya Pathak, Sarah Perrin, André Susano Pinto, Etienne Pot, Angéline Pouget, Alexandre Ramé, Sabela Ramos, Douglas Reid, David Rim, Morgane Rivière, Karsten Roth, Louis Rouillard, Omar Sanseviero, Pier Giuseppe Sessa, Shane Settle, Danila Sinopalnikov, Sara Smoot, Piotr Stanczyk, Andreas Steiner, Lawrence Stewart, Ilya O. Tolstikhin, Michael Tschannen, Anton Tsitsulin, Nino Vieillard, Renjie Wu, Pingmei Xu, Haichuan Yang, Edouard Yvinec, Biao Zhang, Li Zhang, Joe Zou, Nicolas Aagnes, Abdelrahman Abdelhamed, Jakub Adámek, Shivani Agrawal, Shubham Agrawal, Ibrahim Alabdulmohsin, Jean-Baptiste Alayrac, Uri Alon, Chandramouli Amarnath, Ankesh Anand, Chrysovalantis Anastasiou, Setareh Ariafar, François-Xavier Aubet, Kyriakos Axiotis, Federico Barbero, Joelle K. Barral, Alexei Bendebury, Urs Bergmann, Stanley Bileschi, Kat Black, Mathieu Blondel, Sebastian Borgeaud, and Arthur Brazinskas. Gemma 4 technical report. *CoRR*, abs/2607.02770, 2026. doi: 10.48550/ARXIV.2607.02770. URL `https://doi.org/10.48550/arXiv.2607.02770`.

Loubna Ben Allal, Anton Lozhkov, Elie Bakouch, Gabriel Martín Blázquez, Guilherme Penedo, Lewis Tunstall, Andrés Marafioti, Hynek Kydlícek, Agustín Piqueres Lajarín, Vaibhav Srivastav, Joshua Lochner, Caleb Fahlgren, Xuan-Son Nguyen, Clémentine Fourrier, Ben Burtenshaw, Hugo

Larcher, Haojun Zhao, Cyril Zakka, Mathieu Morlon, Colin Raffel, Leandro von Werra, and Thomas Wolf. Smollm2: When smol goes big - data-centric training of a small language model. *CoRR*, abs/2502.02737, 2025. doi: 10.48550/ARXIV.2502.02737. URL https://doi.org/10.48550/arXiv.2502.02737.

AMD ROCm. Dropless MoE training in MaxText on AMD GPUs, 2026. URL https://rocm.blogs.amd.com/software-tools-optimization/maxtext-dropless-moe/README.html. Reports the plain ragged_dot expert matmul as memory-prohibitive at training scale.

Walter Baur and Volker Strassen. The complexity of partial derivatives. *Theoretical Computer Science*, 22(3):317–330, 1983. doi: 10.1016/0304-3975(83)90110-X.

Atilim Gunes Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey. *J. Mach. Learn. Res.*, 18:153:1–153:43, 2017. URL https://jmlr.org/papers/v18/17-468.html.

Heiko Becker, Nikita Zyuzin, Raphaël Monat, Eva Darulova, Magnus O. Myreen, and Anthony C. J. Fox. A verified certificate checker for finite-precision error bounds in coq and HOL4. In Nikolaj S. Bjørner and Arie Gurfinkel (eds.), *2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018*, pp. 1–10. IEEE, 2018. doi: 10.23919/FMCAD.2018.8603019. URL https://doi.org/10.23919/FMCAD.2018.8603019.

Beneficial AI Foundation. FloatSpec. Source repository, 2026. URL https://github.com/Beneficial-AI-Foundation/FloatSpec.

Alexander Bentkamp, Ramon Fernández Mir, and Jeremy Avigad. Verified reductions for optimization. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 74–92. Springer, 2023.

Ulrich Berger and Helmut Schwichtenberg. An inverse of the evaluation functional for typed lambda-calculus. In *Proceedings of the Sixth Annual Symposium on Logic in Computer Science (LICS '91), Amsterdam, The Netherlands, July 15-18, 1991*, pp. 203–211. IEEE Computer Society, 1991. doi: 10.1109/LICS.1991.151645. URL https://doi.org/10.1109/LICS.1991.151645.

Gilbert Bernstein, Michael Mara, Tzu-Mao Li, Dougal Maclaurin, and Jonathan Ragan-Kelley. Differentiating a tensor language. *CoRR*, abs/2008.11256, 2020. URL https://arxiv.org/abs/2008.11256.

Sylvie Boldo and Guillaume Melquiond. Flocq: A unified library for proving floating-point algorithms in coq. In Elisardo Antelo, David Hough, and Paolo Ienne (eds.), *20th IEEE Symposium on Computer Arithmetic, ARITH 2011, Tübingen, Germany, 25-27 July 2011*, pp. 243–252. IEEE Computer Society, 2011. doi: 10.1109/ARITH.2011.40. URL https://doi.org/10.1109/ARITH.2011.40.

James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Yash Katariya, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018. URL https://github.com/jax-ml/jax.

Lukas Bulwahn, Alexander Krauss, Florian Haftmann, Levent Erkök, and John Matthews. Imperative functional programming with Isabelle/HOL. In *Theorem Proving in Higher Order Logics*, volume 5170 of *Lecture Notes in Computer Science*, pp. 134–149. Springer, 2008. doi: 10.1007/978-3-540-71067-7_14.

Tung-Che Chang, Sehyeok Park, Jay P. Lim, and Santosh Nagarakatte. Flops: Semantics, operations, and properties of P3109 floating-point representations in lean. *CoRR*, abs/2602.15965, 2026. doi: 10.48550/ARXIV.2602.15965. URL https://doi.org/10.48550/arXiv.2602.15965.

Junjie Chen, Yihua Liang, Qingchao Shen, Jiajun Jiang, and Shuochuan Li. Toward understanding deep learning framework bugs. *ACM Trans. Softw. Eng. Methodol.*, 32(6):135:1–135:31, 2023. doi: 10.1145/3587155. URL https://doi.org/10.1145/3587155.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training verifiers to solve math word problems. *CoRR*, abs/2110.14168, 2021. URL https://arxiv.org/abs/2110.14168.

Ronan Collobert, Koray Kavukcuoglu, and Clément Farabet. Torch7: A Matlab-like environment for machine learning. In *NeurIPS Workshop on Big Learning*, 2011. URL https://ronan.collobert.com/pub/2011_torch7_nipsw.pdf.

Nasrine Damouche and Matthieu Martel. Salsa: An automatic tool to improve the numerical accuracy of programs. In Bruno Dutertre and Natarajan Shankar (eds.), *Automated Formal Methods, AFM@NFM 2017, Moffett Field, CA, USA, May 19-20, 2017*, volume 5 of *Kalpa Publications in Computing*, pp. 63–76. EasyChair, 2017. doi: 10.29007/J2FD. URL https://doi.org/10.29007/j2fd.

Olivier Danvy. Type-directed partial evaluation. In Hans-Juergen Boehm and Guy L. Steele Jr. (eds.), *Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, St. Petersburg Beach, Florida, USA, January 21-24, 1996*, pp. 242–257. ACM Press, 1996. doi: 10.1145/237721.237784. URL https://doi.org/10.1145/237721.237784.

Eva Darulova, Anastasiia Izycheva, Fariha Nasir, Fabian Ritter, Heiko Becker, and Robert Bastian. Daisy - framework for analysis and optimization of numerical programs (tool paper). In Dirk Beyer and Marieke Huisman (eds.), *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part I*, volume 10805 of *Lecture Notes in Computer Science*, pp. 270–287. Springer, 2018. doi: 10.1007/978-3-319-89960-2_15. URL https://doi.org/10.1007/978-3-319-89960-2_15.

Florent de Dinechin, Christoph Quirin Lauter, and Guillaume Melquiond. Assisted verification of elementary functions using gappa. In Hisham Haddad (ed.), *Proceedings of the 2006 ACM Symposium on Applied Computing (SAC), Dijon, France, April 23-27, 2006*, pp. 1318–1322. ACM, 2006. doi: 10.1145/1141277.1141584. URL https://doi.org/10.1145/1141277.1141584.

Leonardo de Moura and Sebastian Ullrich. The Lean 4 theorem prover and programming language. In André Platzer and Geoff Sutcliffe (eds.), *Automated Deduction – CADE 28*, volume 12699 of *Lecture Notes in Computer Science*, pp. 625–635. Springer, 2021. doi: 10.1007/978-3-030-79876-5_37.

Paulo Emílio de Vilhena and François Pottier. Verifying an effect-handler-based define-by-run reverse-mode AD library. *Log. Methods Comput. Sci.*, 19(4), 2023. doi: 10.46298/LMCS-19(4:5)2023. URL https://doi.org/10.46298/lmcs-19(4:5)2023.

dew contributors. Replacing XLA's ragged-dot lowering with Pallas mixture-of-experts kernels, 2026. URL https://github.com/AshishKumar4/dew/issues/16. AshishKumar4/dew issue #16.

Shizhe Diao, Yu Yang, Yonggan Fu, Xin Dong, Dan Su, Markus Kliegl, Zijia Chen, Peter Belcak, Yoshi Suhara, Hongxu Yin, Mostofa Patwary, Yingyan (Celine) Lin, Jan Kautz, and Pavlo Molchanov. Nemotron-climb: Clustering-based iterative data mixture bootstrapping for language model pre-training. In Danielle Belgrave, Cheng Zhang, Laura N. Montoya, Hsuan-Tien Lin, Razvan Pascanu, Piotr Koniusz, Marzyeh Ghassemi, Nancy Chen, Iván Vladimir Meza Ruíz, and Arturo Loaiza-Bonilla (eds.), *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2025, NeurIPS 2025, San Diego, CA, USA, December 2-7, 2025 / Mexico City, Mexico, November 30 - December 5, 2025*, 2025. URL http://papers.nips.cc/paper_files/paper/2025/hash/24a8968affe71ffe4067d022b9d16566-Abstract-Datasets_and_Benchmarks_Track.html.

Rüdiger Ehlers. Formal verification of piece-wise linear feed-forward neural networks. In Deepak D'Souza and K. Narayan Kumar (eds.), *Automated Technology for Verification and Analysis - 15th International Symposium, ATVA 2017, Pune, India, October 3-6, 2017, Proceedings*, volume 10482 of *Lecture Notes in Computer Science*, pp. 269–286. Springer, 2017. doi: 10.1007/978-3-319-68167-2_19. URL https://doi.org/10.1007/978-3-319-68167-2_19.

Andrei P. Ershov. On programming of arithmetic operations. *Commun. ACM*, 1(8):3–9, 1958. doi: 10.1145/368892.368907. URL https://doi.org/10.1145/368892.368907.

Jean-Christophe Filliâtre and Sylvain Conchon. Type-safe modular hash-consing. In Andrew Kennedy and François Pottier (eds.), *Proceedings of the ACM Workshop on ML, 2006, Portland, Oregon, USA, September 16, 2006*, pp. 12–19. ACM, 2006. doi: 10.1145/1159876.1159880. URL https://doi.org/10.1145/1159876.1159880.

Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, pp. 237–247. ACM, 1993. doi: 10.1145/155090.155113.

Franz Franchetti, Tze Meng Low, Doru-Thom Popovici, Richard Michael Veras, Daniele G. Spampinato, Jeremy R. Johnson, Markus Püschel, James C. Hoe, and José M. F. Moura. SPIRAL: extreme performance portability. *Proc. IEEE*, 106(11):1935–1968, 2018. doi: 10.1109/JPROC.2018.2873289. URL https://doi.org/10.1109/JPROC.2018.2873289.

Roy Frostig, Matthew Johnson, and Chris Leary. Compiling machine learning programs via high-level tracing. 2018. URL https://mlsys.org/Conferences/doc/2018/146.pdf.

Roy Frostig, Matthew J. Johnson, Dougal Maclaurin, Adam Paszke, and Alexey Radul. Decomposing reverse-mode automatic differentiation. *CoRR*, abs/2105.09469, 2021. URL https://arxiv.org/abs/2105.09469.

Robert Joseph George, Will Adkisson, and Anima Anandkumar. Floatlib: Verified floating-point arithmetic in lean, 2026a. URL https://arxiv.org/abs/2609.19352.

Robert Joseph George, Jennifer Cruden, Will Adkisson, Xiangru Zhong, Huan Zhang, and Anima Anandkumar. Torchlean: Formalizing neural networks in lean, 2026b. URL https://arxiv.org/abs/2602.22631.

Jean-Yves Girard. Linear logic. *Theor. Comput. Sci.*, 50:1–102, 1987. doi: 10.1016/0304-3975(87)90045-4. URL https://doi.org/10.1016/0304-3975(87)90045-4.

Giulia Giusti and Michele Pagani. JAX autodiff from a linear logic perspective. *Proc. ACM Program. Lang.*, 10(POPL):324–353, 2026. doi: 10.1145/3776654. URL https://doi.org/10.1145/3776654.

David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, 23(1):5–48, 1991. doi: 10.1145/103162.103163. URL https://doi.org/10.1145/103162.103163.

Eiichi Goto. Monocopy and associative algorithms in an extended LISP. Technical Report TR 74-03, Information Science Laboratory, Faculty of Science, University of Tokyo, April 1974. URL https://www.cs.utexas.edu/~hunt/research/hash-cons/hash-cons-papers/monocopy-goto.pdf.

Andreas Griewank. Who invented the reverse mode of differentiation? *Documenta Mathematica*, Extra Volume: Optimization Stories:389–400, 2012. URL https://www.emis.de/journals/DMJDMV/vol-ismp/52_griewank-andreas-b.html. Extra Volume ISMP.

Andreas Griewank and Shawn Reese. On the calculation of jacobian matrices by the markowitz rule. In Andreas Griewank and George F. Corliss (eds.), *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, pp. 126–135. SIAM, Philadelphia, PA, 1991.

Jiazhen Gu, Xuchuan Luo, Yangfan Zhou, and Xin Wang. Muffin: Testing deep learning libraries via neural architecture fuzzing. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*, pp. 1418–1430. ACM, 2022. doi: 10.1145/3510003.3510092. URL https://doi.org/10.1145/3510003.3510092.

Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, 2020. doi: 10.1038/s41586-020-2649-2.

Les Hatton and Andy Roberts. How accurate is scientific software? *IEEE Trans. Software Eng.*, 20(10):785–797, 1994. doi: 10.1109/32.328993. URL https://doi.org/10.1109/32.328993.

Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. Measuring massive multitask language understanding. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021. URL https://openreview.net/forum?id=d7KBjmI3GmQ.

Daniel Hook and Diane Kelly. Testing for trustworthiness in scientific software. In *ICSE Workshop on Software Engineering for Computational Science and Engineering, SE-CSE 2009, Vancouver, BC, Canada, May 23, 2009*, pp. 59–64. IEEE Computer Society, 2009. doi: 10.1109/SECSE.2009.5069163. URL https://doi.org/10.1109/SECSE.2009.5069163.

Edward J Hu, yelong shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations*, 2022. URL https://openreview.net/forum?id=nZeVKeeFYf9.

Xiaowei Huang, Marta Kwiatkowska, Sen Wang, and Min Wu. Safety verification of deep neural networks. In Rupak Majumdar and Viktor Kuncak (eds.), *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I*, volume 10426 of *Lecture Notes in Computer Science*, pp. 3–29. Springer, 2017. doi: 10.1007/978-3-319-63387-9_1. URL https://doi.org/10.1007/978-3-319-63387-9_1.

Nargiz Humbatova, Gunel Jahangirova, Gabriele Bavota, Vincenzo Riccio, Andrea Stocco, and Paolo Tonella. Taxonomy of real faults in deep learning systems. In Gregg Rothermel and Doo-Hwan Bae (eds.), *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, pp. 1110–1121. ACM, 2020. doi: 10.1145/3377811.3380395. URL https://doi.org/10.1145/3377811.3380395.

Alex Raymond Hyman, Filip Smola, Mark Chevallier, and Jacques D. Fleuriot. IsaGrad: Verified automatic differentiation over computational graphs in imperative HOL. In *Proceedings of the FLOC 2026 Workshop on Logical Methods for Neural Network Analysis*, Lisbon, Portugal, 2026.

JAX contributors. Opt-in GPU Pallas/Triton lowering for ragged_dot fails: the imported module is absent from the released wheels, 2026. URL https://github.com/jax-ml/jax/issues/40888. jax-ml/jax issue #40888; affects releases 0.10.0, 0.10.2, 0.11.1 and 0.11.2.

Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014. URL https://arxiv.org/abs/1408.5093.

Keller Jordan, Yuchen Jin, Vlado Boza, Jiacheng You, Franz Cesista, Laker Newhouse, and Jeremy Bernstein. Muon: An optimizer for hidden layers in neural networks, 2024. URL https://kellerjordan.github.io/posts/muon/.

Upulee Kanewala and James M. Bieman. Techniques for testing scientific programs without an oracle. In Jeffrey C. Carver (ed.), *Proceedings of the 5th International Workshop on Software*

Engineering for Computational Science and Engineering, SE-CSE 2013, San Francisco, California, USA, May 18, 2013, pp. 48–57. IEEE Computer Society, 2013. doi: 10.1109/SECSE.2013.6615099. URL https://doi.org/10.1109/SECSE.2013.6615099.

Andrej Karpathy. nanochat: The best chatgpt that $100 can buy, 2025. URL https://github.com/karpathy/nanochat.

Guy Katz, Clark W. Barrett, David L. Dill, Kyle Julian, and Mykel J. Kochenderfer. Reluplex: An efficient SMT solver for verifying deep neural networks. In Rupak Majumdar and Viktor Kuncak (eds.), *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I*, volume 10426 of *Lecture Notes in Computer Science*, pp. 97–117. Springer, 2017. doi: 10.1007/978-3-319-63387-9_5. URL https://doi.org/10.1007/978-3-319-63387-9_5.

Guy Katz, Derek A. Huang, Duligur Ibeling, Kyle Julian, Christopher Lazarus, Rachel Lim, Parth Shah, Shantanu Thakoor, Haoze Wu, Aleksandar Zeljic, David L. Dill, Mykel J. Kochenderfer, and Clark W. Barrett. The marabou framework for verification and analysis of deep neural networks. In Isil Dillig and Serdar Tasiran (eds.), *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I*, volume 11561 of *Lecture Notes in Computer Science*, pp. 443–452. Springer, 2019. doi: 10.1007/978-3-030-25540-4_26. URL https://doi.org/10.1007/978-3-030-25540-4_26.

Konstantin Kaulen, Tobias Ladner, Stanley Bak, Christopher Brix, Hai Duong, Thomas Flinkow, Taylor T. Johnson, Lukas Koller, Edoardo Manino, ThanhVu H. Nguyen, and Haoze Wu. The 6th international verification of neural networks competition (VNN-COMP 2025): Summary and results. *CoRR*, abs/2512.19007, 2025. doi: 10.48550/ARXIV.2512.19007. URL https://doi.org/10.48550/arXiv.2512.19007.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In Yoshua Bengio and Yann LeCun (eds.), *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015. URL http://arxiv.org/abs/1412.6980.

Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 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.

John Launchbury and Simon L. Peyton Jones. Lazy functional state threads. In Vivek Sarkar, Barbara G. Ryder, and Mary Lou Soffa (eds.), *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI), Orlando, Florida, USA, June 20-24, 1994*, pp. 24–35. ACM, 1994. doi: 10.1145/178243.178246. URL https://doi.org/10.1145/178243.178246.

Augustin Lemesle, Julien Lehmann, and Tristan Le Gall. Neural network verification with pyrat, 2024. URL https://arxiv.org/abs/2410.23903.

Chenyi Li, Ziyu Wang, Wanyi He, Yuxuan Wu, Shengyang Xu, and Zaiwen Wen. Formalization of convergence rates of four first-order algorithms for convex optimization. *J. Autom. Reason.*, 69(4):28, 2025. doi: 10.1007/S10817-025-09741-W. URL https://doi.org/10.1007/s10817-025-09741-w.

Seppo Linnainmaa. Taylor expansion of the accumulated rounding error. *BIT Computer Science and Numerical Mathematics*, 16(2):146–160, 1976. doi: 10.1007/BF01931367. URL https://doi.org.

Jiawei Liu, Jinkun Lin, Fabian Ruffy, Cheng Tan, Jinyang Li, Aurojit Panda, and Lingming Zhang. Nnsmith: Generating diverse and valid test cases for deep learning compilers. In Tor M. Aamodt, Natalie D. Enright Jerger, and Michael M. Swift (eds.), *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*, pp. 530–543. ACM,

2023a. doi: 10.1145/3575693.3575707. URL https://doi.org/10.1145/3575693.3575707.

Jiawei Liu, Jinjun Peng, Yuyao Wang, and Lingming Zhang. Neuri: Diversifying DNN generation via inductive rule inference. In Satish Chandra, Kelly Blincoe, and Paolo Tonella (eds.), *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*, pp. 657–669. ACM, 2023b. doi: 10.1145/3611643.3616337. URL https://doi.org/10.1145/3611643.3616337.

LMSYS Org. SGLang-Jax, 2025. URL https://www.lmsys.org/blog/2025-10-29-sglang-jax/. Replaces the ragged_dot-based expert matmul with a Megablox grouped-matmul operator; the reported speedup is measured on TPU.

Alessio Lomuscio and Lalit Maganti. An approach to reachability analysis for feed-forward relu neural networks. *CoRR*, abs/1706.07351, 2017. URL http://arxiv.org/abs/1706.07351.

Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019. URL https://openreview.net/forum?id=Bkg6RiCqY7.

Dougal Maclaurin, David Duvenaud, and Ryan P. Adams. Autograd: Effortless gradients in NumPy. In *ICML 2015 AutoML Workshop*, 2015. URL https://indico.ijclab.in2p3.fr/event/2914/contributions/6483/subcontributions/180/attachments/6060/7185/automl-short.pdf.

Dougal Maclaurin, Alexey Radul, Matthew J. Johnson, , and Dimitrios Vytiniotis. Dex: array programming with typed indices. In *Program Transformations for ML Workshop at NeurIPS 2019*, 2019. URL https://openreview.net/forum?id=rJxd7vsWPS.

Victor Magron, George A. Constantinides, and Alastair F. Donaldson. Certified roundoff error bounds using semidefinite programming. *ACM Trans. Math. Softw.*, 43(4):34:1–34:31, 2017. doi: 10.1145/3015465. URL https://doi.org/10.1145/3015465.

MaxText contributors. MaxText mixture-of-experts configuration, 2026. URL https://maxtext.readthedocs.io/en/latest/reference/core_concepts/moe_configuration.html.

Mariano M. Moscato, Laura Titolo, Aaron Dutle, and César A. Muñoz. Automatic estimation of verified floating-point round-off errors via static analysis. In Stefano Tonetta, Erwin Schoitsch, and Friedemann Bitsch (eds.), *Computer Safety, Reliability, and Security - 36th International Conference, SAFECOMP 2017, Trento, Italy, September 13-15, 2017, Proceedings*, volume 10488 of *Lecture Notes in Computer Science*, pp. 213–229. Springer, 2017. doi: 10.1007/978-3-319-66266-4_14. URL https://doi.org/10.1007/978-3-319-66266-4_14.

Uwe Naumann. Optimal jacobian accumulation is np-complete. *Math. Program.*, 112(2):427–441, 2008. doi: 10.1007/S10107-006-0042-Z. URL https://doi.org/10.1007/s10107-006-0042-z.

NVIDIA. Accelerating dropless MoE training in JAX with NVIDIA Transformer Engine, 2026. URL https://developer.nvidia.com/blog/accelerating-dropless-moe-training-in-jax-with-nvidia-transformer-engine.

OpenXLA. Ragged operations in CHLO, 2025. URL https://groups.google.com/a/openxla.org/g/openxla-discuss/c/wmb1OpnkUd0. openxla-discuss announcement.

OpenXLA contributors. Add a Triton kernel for ragged-dot on GPU, 2026. URL https://github.com/openxla/xla/pull/45998. Pull request series openxla/xla #45998 and #46001. As of August 2026 the grouped kernel had not shipped in a released JAX.

OpenXLA Project. XLA: A machine learning compiler for GPUs, CPUs, and ML accelerators. https://github.com/openxla/xla, 2026. Accessed September 2026.

Pavel Panchekha, Alex Sanchez-Stern, James R. Wilcox, and Zachary Tatlock. Automatically improving accuracy for floating point expressions. In David Grove and Stephen M. Blackburn (eds.), *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pp. 1–11. ACM, 2015. doi: 10.1145/2737924.2737959. URL https://doi.org/10.1145/2737924.2737959.

Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in PyTorch. In *NeurIPS 2017 Workshop on Autodiff*, 2017. URL https://openreview.net/forum?id=BJJsrmfCZ.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, volume 32, 2019. URL https://proceedings.neurips.cc/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html.

Adam Paszke, Daniel D. Johnson, David Duvenaud, Dimitrios Vytiniotis, Alexey Radul, Matthew J. Johnson, Jonathan Ragan-Kelley, and Dougal Maclaurin. Getting to the point: index sets and parallelism-preserving autodiff for pointful array programming. *Proc. ACM Program. Lang.*, 5 (ICFP):1–29, 2021. doi: 10.1145/3473593. URL https://doi.org/10.1145/3473593.

Zedong Peng, Xuanyi Lin, Michelle Simon, and Nan Niu. Unit and regression tests of scientific software: A study on SWMM. *J. Comput. Sci.*, 53:101347, 2021. doi: 10.1016/J.JOCS.2021.101347. URL https://doi.org/10.1016/j.jocs.2021.101347.

Hung Viet Pham, Thibaud Lutellier, Weizhen Qi, and Lin Tan. CRADLE: cross-backend validation to detect and localize bugs in deep learning libraries. In Joanne M. Atlee, Tevfik Bultan, and Jon Whittle (eds.), *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, pp. 1027–1038. IEEE / ACM, 2019. doi: 10.1109/ICSE.2019.00107. URL https://doi.org/10.1109/ICSE.2019.00107.

Qwen Team. Qwen3.8-Max: A new bar for coding and cowork, August 2026. URL https://qwen.ai/blog?id=qwen3.8.

Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. 2019. URL https://api.semanticscholar.org/CorpusID:160025533.

Alexey Radul, Adam Paszke, Roy Frostig, Matthew J. Johnson, and Dougal Maclaurin. You only linearize once: Tangents transpose to gradients. *Proc. ACM Program. Lang.*, 7(POPL):1246–1274, 2023. doi: 10.1145/3571236. URL https://doi.org/10.1145/3571236.

rdyro. A Pallas GPU kernel for ragged_dot, 2026. URL https://github.com/rdyro/gpu_ragged_dot.

Sashank J. Reddi, Satyen Kale, and Sanjiv Kumar. On the convergence of adam and beyond. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018. URL https://openreview.net/forum?id=ryQu7f-RZ.

Talia Ringer, Karl Palmskog, Ilya Sergey, Milos Gligoric, and Zachary Tatlock. QED at large: A survey of engineering of formally verified software. *Found. Trends Program. Lang.*, 5(2-3):102–281, 2019. doi: 10.1561/2500000045. URL https://doi.org/10.1561/2500000045.

David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986. doi: 10.1038/323533a0.

Amit Sabne. Xla : Compiling machine learning for peak performance, 2020.

Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic superoptimization. In Vivek Sarkar and Rastislav Bodík (eds.), *Architectural Support for Programming Languages and Operating Systems, ASPLOS 2013, Houston, TX, USA, March 16-20, 2013*, pp. 305–316. ACM, 2013. doi: 10.1145/2451116.2451150. URL https://doi.org/10.1145/2451116.2451150.

Daniel Selsam, Percy Liang, and David L. Dill. Developing bug-free machine learning systems with formal mathematics. In Doina Precup and Yee Whye Teh (eds.), *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, volume 70 of *Proceedings of Machine Learning Research*, pp. 3047–3056. PMLR, 2017. URL http://proceedings.mlr.press/v70/selsam17a.html.

Daniel Selsam, Simon Hudon, and Leonardo de Moura. Sealing pointer-based optimizations behind pure functions. *Proc. ACM Program. Lang.*, 4(ICFP):115:1–115:20, 2020. doi: 10.1145/3408997. URL https://doi.org/10.1145/3408997.

Jeffrey Mark Siskind and Barak A. Pearlmutter. Perturbation confusion and referential transparency: Correct functional implementation of forward-mode AD. In *Implementation and Application of Functional Languages: 17th International Workshop, IFL 2005*, pp. 1–9, Dublin, Ireland, 2005. Trinity College Dublin Computer Science Department Technical Report TCD-CS-2005-60.

Jeffrey Mark Siskind and Barak A. Pearlmutter. Nesting forward-mode AD in a functional framework. *Higher-Order and Symbolic Computation*, 21(4):361–376, 2008. doi: 10.1007/s10990-008-9037-1.

Tomáš Skřivan. SciLean: Scientific computing in Lean, 2024. URL https://github.com/lecopivo/SciLean. GitHub repository, version v4.20.1.

Alexey Solovyev, Marek S. Baranowski, Ian Briggs, Charles Jacobsen, Zvonimir Rakamaric, and Ganesh Gopalakrishnan. Rigorous estimation of floating-point round-off errors with symbolic taylor expansions. *ACM Trans. Program. Lang. Syst.*, 41(1):2:1–2:39, 2019. doi: 10.1145/3230733. URL https://doi.org/10.1145/3230733.

Mirko Spasic and Filip Maric. Formalization of incremental simplex algorithm by stepwise refinement. In Dimitra Giannakopoulou and Dominique Méry (eds.), *FM 2012: Formal Methods - 18th International Symposium, Paris, France, August 27-31, 2012. Proceedings*, volume 7436 of *Lecture Notes in Computer Science*, pp. 434–449. Springer, 2012. doi: 10.1007/978-3-642-32759-9_35. URL https://doi.org/10.1007/978-3-642-32759-9_35.

Bert Speelpenning. *Compiling Fast Partial Derivatives of Functions Given by Algorithms*. PhD thesis, University of Illinois Urbana-Champaign, USA, 1980. URL https://hdl.handle.net/2142/66437.

Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. Secure distributed programming with value-dependent types. *J. Funct. Program.*, 23(4):402–451, 2013. doi: 10.1017/S0956796813000142. URL https://doi.org/10.1017/S0956796813000142.

The mathlib Community. The Lean mathematical library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP 2020)*, pp. 367–381. ACM, 2020. doi: 10.1145/3372885.3373824.

The Plausible contributors. Plausible: A property testing framework for Lean 4. https://github.com/leanprover-community/plausible. Accessed September 24, 2026.

The Theano Development Team. Theano: A Python framework for fast computation of mathematical expressions. *arXiv preprint arXiv:1605.02688*, 2016. URL https://arxiv.org/abs/1605.02688.

Vincent Tjeng, Kai Yuanqing Xiao, and Russ Tedrake. Evaluating robustness of neural networks with mixed integer programming. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019. URL https://openreview.net/forum?id=HyGIdiRqtm.

Fei Wang, Daniel Zheng, James M. Decker, Xilun Wu, Grégory M. Essertel, and Tiark Rompf. Demystifying differentiable programming: shift/reset the penultimate backpropagator. *Proc. ACM Program. Lang.*, 3(ICFP):96:1–96:31, 2019. doi: 10.1145/3341700. URL https://doi.org/10.1145/3341700.

Gan Wang, Zan Wang, Junjie Chen, Xiang Chen, and Ming Yan. An empirical study on numerical bugs in deep learning programs. In *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022*, pp. 173:1–173:5. ACM, 2022. doi: 10.1145/3551349.3559561. URL https://doi.org/10.1145/3551349.3559561.

Anjiang Wei, Yinlin Deng, Chenyuan Yang, and Lingming Zhang. Free lunch for testing: Fuzzing deep-learning libraries from open source. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*, pp. 995–1007. ACM, 2022. doi: 10.1145/3510003.3510041. URL https://doi.org/10.1145/3510003.3510041.

R. E. Wengert. A simple automatic derivative evaluation program. *Commun. ACM*, 7(8):463–464, 1964. doi: 10.1145/355586.364791. URL https://doi.org/10.1145/355586.364791.

P. J. Werbos. *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*. PhD thesis, Harvard University, 1974.

James Hardy Wilkinson. Rounding errors in algebraic processes. In *Information Processing, Proceedings of the 1st International Conference on Information Processing, UNESCO, Paris 15-20 June 1959*, pp. 44–53. UNESCO (Paris), 1959.

Vadim Zaliva. *HELIX: From Math to Verified Code*. PhD thesis, Carnegie Mellon University, USA, 2021. URL https://doi.org/10.1184/r1/13636808.v1.

Huan Zhang, Tsui-Wei Weng, Pin-Yu Chen, Cho-Jui Hsieh, and Luca Daniel. Efficient neural network robustness certification with general activation functions. *Advances in Neural Information Processing Systems*, 31:4939–4948, 2018. URL https://arxiv.org/pdf/1811.00866.pdf.

A Appendix

A.1 Extended Related Work

Relationship with Certigrad.

- Certigrad focuses on the formalization of stochastic backpropagation, whereas we focus on general automatic differentiation and do not formalize stochastic backpropagation.
- Certigrad axiomitized probability theory and calculus, whereas we connect with Mathlib's definitions.
- Certigrad implementations run exclusively on CPU.

Relationship with SciLean.

- SciLean runs on CPU, whereas DiffDog supports compilation to run on GPUs and TPUs.
- SciLean implements AD via tactic metaprogramming, whereas we use a tracing-based AD design.
- Some SciLean operation proofs contain `sorry`. DiffDog does not rely on `sorry`.
- SciLean does not support automatic vectorization but instead offers a generic comprehensions programming model that is similar in spirit but does not cause operations to be compiled or run with vector instructions.

Relationship with TorchLean. The implementation of a verified AD and neural network library in Lean is explored in George et al. (2026b). After initial release, CUDA compilation and GPU execution support was added as well. George et al. (2026b) focuses on neural robustness certification via IBP, CROWN, Alpha-Beta-CROWN, and so on. Accordingly, they emphasize configurable numerics, certificate optimizer interop. In contrast, this work focuses on developing a foundational library for numerical programming, machine learning research, and program transformation. We therefore support features like

- TorchLean compiles and links custom `.cu` files at build time producing executables that dispatch kernels on GPU. In contrast, we leverage existing ML compilers like XLA (Sabne, 2020). DIFFDOG emits StableHLO MLIR that is JIT compiled at runtime. In contrast to TorchLean, whole-program compilation gives DIFFDOG programs optimizations like kernel fusion and kernel dispatch based on the exact runtime accelerator and input shapes, and enables DIFFDOG programs to run on other accelerators including TPUs. DIFFDOG links against the PJRT runtime `.so` libraries for dispatching and execution.
- TorchLean supports configurable numerics for model robustness. DIFFDOG supports float32. Support for alternative numerics is planned but out of scope for this presentation. Support for configurable numerics is not planned but conceivably possible.
- TorchLean implements a tape-based AD design, with tape state being tracked in a state monad. Therefore, user programs are required to follow a monadic programming model. In contrast, DIFFDOG supports tracing of plain lean functions and expressions without requiring monadic programming.
- We support arbitrary nested AD whereas TorchLean is limited to Hessian-vector products.
- We support automatic vectorization. TorchLean does not.
- We support extensible op families such that users can extend the system to suit there needs. TorchLean requires users to use a limited set provide by an exhaustive inductive data type.
- TorchLean provides results for neural network verification. We do not.

Relationship with JAX. We deliberately follow the JAX and numpy external APIs as closely as possible in the interest of minimizing user friction. Internally, we differ in our approaches to AD:

- DIFFDOG uses rank-2 types to handle perturbation confusion in nested AD, whereas JAX relies on tracing levels tracked pointer identity of Python `TraceTag` objects.
- DIFFDOG represents models forward and reverse mode transformations separately, whereas JAX represents reverse mode as the transposition of forward mode primitives.
- DIFFDOG utilizes its core IR for all transformations, whereas JAX uses nested interpreters in some cases, e.g. for `vmap`.

JAX decomposes reverse mode AD as the composition of linearization and the transposition of linear Jacobian primitives (Frostig et al., 2021). Radul et al. (2023) and Giusti & Pagani (2026) formalize this model in lambda calculus, and connect the linearization step with Girard's linear logic (Girard, 1987). In contrast to the linearize-then-transpose design, we maintain separate forward and reverse mode rules for each op and provide certificates that they are mutually adjoint.

Also loosely in the JAX lineage, the Dex language (Maclaurin et al., 2019; Paszke et al., 2021) proposes a language purpose-built for array programming, including AD support and value-dependent types for shapes (Swamy et al., 2013). Bernstein et al. (2020) expose a performance pitfall in array-based reverse mode AD, and present a purely functional approach that maintains performance without requiring mutation.

Numerical analysis and neural network verification. Wilkinson (1959); Goldberg (1991) characterize round-off errors in numerical analysis. Boldo & Melquiond (2011); Beneficial AI Foundation (2026); Chang et al. (2026); George et al. (2026a) develop verified libraries for floating point numerics and their analysis. Becker et al. (2018); Solovyev et al. (2019); Magron et al. (2017); Moscato et al. (2017); de Dinechin et al. (2006) offer round-off bound certificates for floating point programs. Panchekha et al. (2015); Damouche & Martel (2017); Darulova et al. (2018); Schkufza

et al. (2013) optimize floating point programs for accuracy and performance. Franchetti et al. (2018); Zaliva (2021) optimize numerical programs with respect to the their real number semantics.

Katz et al. (2017; 2019); Huang et al. (2017); Ehlers (2017); Tjeng et al. (2019); Lomuscio & Maganti (2017) pioneered neural network verification, the certification of neural networks to obey given specifications. The VNN-COMP competition (Kaulen et al., 2025) has stimulated progress in the area. At the time of writing, leading methods include α, β-CROWN (Zhang et al., 2018), VibeCheck, and PyRAT (Lemesle et al., 2024).

A.2 A Detailed Look at the DiffDog Core

The core is defined with extension in mind. We walk through elementwise multiplication and a small program computing $x \odot x + y$.

Shapes and operation signatures. An array shape is a list of dimensions. So `[]` is the shape of a scalar, `[n]` of a vector, and `[m, n]` of a matrix. A signature specifies an operation family indexed by its list of input shapes and its single output shape:

```
abbrev Shape := List Nat
abbrev Sig := List Shape → Shape → Type

inductive MultiplyOp : Sig where
  | multiply : MultiplyOp [s, s] s
```

The constructor `MultiplyOp.multiply` describes elementwise multiplication of two arrays of the same shape. It contains neither the arrays themselves nor an implementation of multiplication. Its type records the operation's arity and shape constraints: two inputs of shape `s` produce one output of shape `s`. The shape is an implicit parameter, so this one constructor covers scalars, vectors, and higher-rank arrays. More generally, an operation constructor can carry static parameters, such as reduction axes, while its indices specify the shapes of its dynamic operands and result.

Giving operations meaning. An interpreter supplies the meaning of an operation for an array family `Arr`. We write `Arr⟦shapes⟧` for a heterogeneous product containing one `Arr s` for each shape `s` in `shapes`. The notation `t![x, y]` constructs such a product; internally it is `(x, (y, PUnit.unit))`.

```
class Interpretable (Op : Sig) (Arr : Shape → Type u) where
  interpret : {inshapes : List Shape} → {outshape : Shape} →
    Op inshapes outshape → Arr⟦inshapes⟧ → Arr outshape

instance [∀ s, Mul (Arr s)] : Interpretable MultiplyOp Arr where
  interpret
    | .multiply, (x, (y, _)) ⇒ x * y
```

This instance interprets multiplication using the array family's existing multiplication operation as provided by `Mul (Arr s)`. In particular, `LeanArray ℝ` gives mathematical, real-valued array semantics. The signature does not change when the interpreter changes: the operation's syntax is separate from its meaning over a particular representation. The equal-shape restriction comes from `MultiplyOp`'s `[s, s]`, not from `Interpretable`. For example, if `inshapes` is `[[m, n], [n, p]]`, then `Arr⟦inshapes⟧` contains an `Arr [m, n]` and an `Arr [n, p]`, as needed for matrix multiplication.

Defining the function users actually call. The function `np.multiply` takes arrays as arguments and returns an array. Restricting it to equal-shaped inputs for this example gives:

```
def np.multiply {Arr : Shape → Type u} {s : Shape}
    [Interpretable MultiplyOp Arr]
```

```
    (x1 x2 : Arr s) : Arr s :=
  Interpretable.interpret MultiplyOp.multiply t![x1, x2]
```

Users write `np.multiply x1 x2`. Lean infers the shape and instance synthesis supplies the interpreter for `Arr`. The function passes the operation and its operands to that interpreter, so the same definition works with any array family that supports multiplication. In short, `MultiplyOp.multiply` describes an operation, `Interpretable` supplies its meaning for a particular array family, and `np.multiply` is the front door ordinary Lean function that users call. Our full implementation of `np.multiply` also supports NumPy-style broadcasting but we skip it here for simplicity.

An extensible operation language. The core does not enumerate a fixed set of primitives in one large inductive type. Instead, independent operation families are combined by a sum:

```
def SumSig (A B : Sig) : Sig :=
  fun inputs output ⇒ A inputs output ⊕ B inputs output

class Inj (Small Big : Sig) where
  inj : {inputs : List Shape} → {output : Shape} →
    Small inputs output → Big inputs output
```

An operation in `SumSig A B` is either an `A` operation or a `B` operation, with the same input and output indices. (Large combinations can exceed Lean's default instance-synthesis budget. We also provide `assembled_sig`, which arranges the operation families into groups and precomputes their instances, rather than asking Lean to assemble the entire set at each use. It retains the same `SumSig` representation.) For example, `Inj.inj` wraps a `MultiplyOp` operation in `.inr` to place it in `SumSig AddOp MultiplyOp`; placing it in `MultiplyOp` itself requires no wrapper. Lean finds the required wrappers automatically, including for nested sums.

For example, `AddOp.add : AddOp [s, s] s` describes homogeneous addition. Our example program needs only these two families:

```
abbrev ExampleSig := SumSig MultiplyOp AddOp

instance [Interpretable A Arr] [Interpretable B Arr] :
    Interpretable (SumSig A B) Arr where
  interpret
    | .inl op, args ⇒ Interpretable.interpret op args
    | .inr op, args ⇒ Interpretable.interpret op args
```

The interpreter for a combined signature simply dispatches to the component interpreters. A user-defined family can be added by forming another sum and supplying its instances, without changing the core program representation or evaluator. `np.Sig` provides an assembly of all numpy operations, and is not a privileged feature or built into the IR. Injection describes how to embed syntax, and Lean automatically synthesizes the injection path for us directed by types.

Contexts and typed references. To connect operations into programs, we need references to their inputs and intermediate results. A context records the shape of every available variable, with the most recently bound variable first:

```
abbrev Ctx := List Shape

def VarAt (s : Shape) : Ctx → Nat → Prop
  | [], _ ⇒ False
  | s' :: _, 0 ⇒ s' = s
  | _ :: Γ, n + 1 ⇒ VarAt s Γ n

def Var (Γ : Ctx) (s : Shape) : Type := { n : Nat // VarAt s Γ n }
```

```
def Var.vz {Γ : Ctx} {s : Shape} : Var (s :: Γ) s := ⟨0, rfl⟩

def Var.vs {Γ : Ctx} {s s' : Shape} (v : Var Γ s) :
    Var (s' :: Γ) s := ⟨v.1.succ, v.2⟩
```

A `Var Γ s` is a de Bruijn index paired with a proof that the referenced entry exists and has shape `s`. The proof is erased during execution. `Var.vz` refers to the newest entry; `Var.vs` shifts an existing reference past a newly added entry. For example, in a context holding `x` followed by `y`, `.vz` refers to `x` and `.vs .vz` to `y`. Extending the context changes indices but not the values they designate.

Instructions and straight-line bodies. An instruction packages an operation with references to its operands:

```
structure Instr (Op : Sig) (Γ : Ctx) (outshape : Shape) where
  inshapes : List Shape
  op : Op inshapes outshape
  args : (Var Γ)⟦inshapes⟧

inductive ANFSeq (Op : Sig) : Ctx → Ctx → Type where
  | nil : ANFSeq Op Γ Γ
  | snoc : ANFSeq Op Γ₀ Γ → Instr Op Γ output →
      ANFSeq Op Γ₀ (output :: Γ)
```

The shape indices connect the three instruction fields: each argument must refer to an available variable of the shape required by the operation. An instruction cannot refer to a value that has not yet been bound.

An `ANFSeq Op Γ₀ Γ` is a straight-line sequence of let-bindings that extends input context `Γ₀` to context `Γ`. The empty sequence leaves the context unchanged. Each `snoc` appends an instruction to the execution order and prepends its result's shape to the context. This is A-normal form: an instruction takes variables as operands, not nested instruction expressions. Sharing is explicit because several operands can reference the same variable. We do not yet support conditionals or loops, so there is currently no material distinction between A-normal form and static single assignment (SSA) for our purposes. We foresee adopting some form of loops and conditionals with an ANF approach in the future.

From a body to a function. An `ANFSeq` accumulates all intermediate values, but does not say which ones to return. `ANFFun` adds that boundary:

```
structure ANFFun (Op : Sig) (Γin : Ctx) (outs : List Shape) where
  Γout : Ctx
  body : ANFSeq Op Γin Γout
  outvars : (Var Γout)⟦outs⟧
```

The internal context `Γout` is stored in the function rather than exposed in its type. Consequently, transformations can change the number of intermediate bindings while retaining a fixed input/output interface. The output references may select, reorder, or repeat any variables in the final context. Each primitive has one output, but an `ANFFun` can return several arrays.

We can now construct $x \odot x + y$ directly. Both inputs and the output have shape `s`; setting `s := []` gives a scalar example.

```
def squarePlus {s : Shape} : ANFFun ExampleSig [s, s] [s] where
  Γout := [s, s, s, s]
  body := .snoc
    (.snoc .nil {
```

```
      inshapes := [s, s]
      op := .inl MultiplyOp.multiply
      args := t![.vz, .vz] })
    { inshapes := [s, s]
      op := .inr AddOp.add
      args := t![.vz, .vs (.vs .vz)] }
  outvars := t![.vz]
```

Of course no one actually programs this way, and instead tracing performs this lowering for us.

Interpreting a complete program. A value environment uses the same heterogeneous-product representation as operation arguments. Evaluation follows the structure of the instruction sequence:

```
abbrev Env (Arr : Shape → Type u) (Γ : Ctx) : Type u := Arr⟦Γ⟧

def ANFSeq.eval [Interpretable Op Arr] :
    ANFSeq Op Γ₀ Γ → Env Arr Γ₀ → Env Arr Γ
  | .nil, ρ ⇒ ρ
  | .snoc p i, ρ ⇒
      let ρ' := ANFSeq.eval p ρ
      (Interpretable.interpret i.op (List.TProd.lookupEnv i.args ρ'), ρ')

def ANFFun.evalSpec [Interpretable Op Arr]
    (g : ANFFun Op Γin outs) (ρ : Env Arr Γin) : Arr⟦outs⟧ :=
  List.TProd.lookupEnv g.outvars (ANFSeq.eval g.body ρ)
```

Here `List.TProd.lookupEnv` replaces each typed variable reference with its value in the environment. An instruction evaluates its prefix, fetches its operands, interprets its operation, and extends the environment with the result. Function evaluation then selects the declared outputs. Under the real-array interpretation, evaluating `squarePlus` on `t![x, y]` returns `t![x * x + y]`.

Authoring transformation rules. Operation extensibility also applies to differentiation. A primitive supplies derivative programs over a host signature `S`:

```
class ADable (Op : Sig) (S : Sig) where
  vjp : {ins : List Shape} → {out : Shape} →
    Op ins out → ANFFun S (out :: ins) ins
  jvp : {ins : List Shape} → {out : Shape} →
    Op ins out → ANFFun S (ins ++ ins) [out]
```

The reverse rule receives the output cotangent followed by the primal inputs, and returns one cotangent for each input. The forward rule receives the primal inputs followed by their tangents, and returns the output tangent. These rules are syntax, not callbacks specialized to an array representation.

We provide a custom `emit%` syntax to ease the construction of ANF fragments for operation authors. For example, the multiplication module defines its reverse rule as follows:

```
def multiplyVjp {S : Sig} [MultiplyOp ⊆ S] {Γ : Ctx} {s : Shape}
    (a b ct : Var Γ s) : ANFFun S Γ [s, s] :=
  emit% do
    let da ← emitOp MultiplyOp.multiply t![b, ct]
    let db ← emitOp MultiplyOp.multiply t![a, ct]
    return t![da, db]
```

Here `⊆` is notation for `Inj`. The two emitted instructions compute $b \odot \bar{c}$ and $a \odot \bar{c}$, the cotangents of the two inputs. `emitOp` injects each primitive into the host signature. The variables `a`, `b`, and

`ct` remain usable after emission without manually inserting `Var.vs`. The forward rule similarly emits $\dot{a} \odot b + a \odot \dot{b}$, so the complete multiplication instance additionally requires addition in its host:

```
instance instADableMultiplyOp {S : Sig} [MultiplyOp ⊆ S] [AddOp ⊆ S] :
    ADable MultiplyOp S where
  vjp
    | .multiply ⇒ multiplyVjp revArgs.hd revArgs.tl.hd revCt
  jvp
    | .multiply ⇒
        multiplyJvp fwdArgs.hd fwdArgs.tl.hd fwdTangents.hd fwdTangents.tl.hd
```

Keeping `Op` and `S` separate lets derivative rules use operations beyond the differentiated primitive's own family. Whole-program reverse mode also requires zero and addition operations for cotangent initialization and accumulation.

Completing an operation's support. A derivative implementation is not itself a correctness proof. For verified AD, operation authors supply `FDerivDomain`, describing the mathematical domain, `FDerivLaw`, connecting the emitted forward rule to the Fréchet derivative, and `AdjointLaw`, relating the forward and reverse rules. These local contracts are used by the general correctness theorems for transformations of ANF programs. Multiplication is defined on all real-array inputs.

Compilation is another per-operation capability. For example, the multiplication module supplies the following StableHLO emission rule:

```
instance : EmitHLO MultiplyOp where
  emit
    | @MultiplyOp.multiply s, (x, (y, _)) ⇒
        pushInstr (.multiply x y) (.f32 s)
```

Because all these type classes compose with SumSig, adding a new primitive consists of defining its syntax and the interpretations, transformation rules, and so on. None of these additions requires changing the `Instr`, `ANFSeq`, or `ANFFun` datatypes.

A.3 (Stochastic) Gradient Descent Convergence Theorems

```
theorem gd_rate (s : Shape)
    (f : EuclideanArray s → ℝ)
    (gradient : EuclideanArray s → EuclideanArray s)
    (x0 xStar : EuclideanArray s) (L gamma : ℝ)
    (hgrad : ∀ z, HasGradientAt f (gradient z) z)
    (hconv : ConvexOn ℝ Set.univ f)
    (hLpos : 0 < L)
    (hLip : LipschitzWith ⟨L, hLpos.le⟩ gradient)
    (hgamma : 0 < gamma) (hgammaLe : gamma ≤ 1 / L)
    (hmin : ∀ z, f xStar ≤ f z)
    (t : ℕ) (ht : 0 < t) :
    f (gradientDescent (EuclideanArray.scalar gamma) gradient x0 t) -
      f xStar ≤ ‖x0 - xStar‖ ^ 2 / (2 * gamma * (t : ℝ))

theorem sgd_rate {n : ℕ} (hn : 0 < n)
    {loss : Fin n → E → ℝ} {grad : Fin n → E → E} {Lmax gamma0 : ℝ}
    (hconv : ∀ i, ConvexOn ℝ Set.univ (loss i))
    (hgrad : ∀ i x, HasGradientAt (loss i) (grad i x) x)
    (hL : 0 < Lmax) (hLip : ∀ i, LipschitzWith ⟨Lmax, hL.le⟩ (grad i))
    {xStar x0 : E} (hmin : ∀ z, objective loss xStar ≤ objective loss z)
    (hgamma0 : 0 < gamma0) (hgamma0Le : gamma0 ≤ 1 / (4 * Lmax))
    (T : ℕ) (hT : 0 < T) :
    (𝔼 ω : Draws n T,
      (objective loss (pathMean x0 (gamma0 / Real.sqrt (T : ℝ)) grad ω) -
        objective loss xStar)) ≤
```

```
      ‖x0 - xStar‖ ^ 2 / (gamma0 * Real.sqrt (T : ℝ)) +
        (2 * gamma0 * gradientNoise loss grad) / Real.sqrt (T : ℝ)
```

A.4 Formalization of the "Cheap Gradient" Principle

We list the core pieces of the formalized cheap gradient theorem here, with complete code in `Experimental/BaurStrassen.lean`.

```
/-- A Sig for arithmetic on arrays. -/
inductive BaurStrassenSig : Sig where
  | add : BaurStrassenSig [s, s] s
  | sub : BaurStrassenSig [s, s] s
  | mul : BaurStrassenSig [s, s] s
  | div : BaurStrassenSig [s, s] s
  | zero : BaurStrassenSig [] s
  | one : BaurStrassenSig [] s

/-- A signature whose reverse rules and primitive arities have a uniform bound. -/
class BoundedADable (Op S : Sig) [ADable Op S] where
  /-- Constant bound on reverse-rule emission plus cotangent accumulation. -/
  vjpSizeBound : Nat
  /-- Every primitive operation fits within `vjpSizeBound`. -/
  vjp_cost_le : ∀ {ins : List Shape} {out : Shape} (op : Op ins out),
    (ADable.vjp (S := S) op).size + ins.length ≤ vjpSizeBound

instance : BoundedADable BaurStrassenSig BaurStrassenSig where
  vjpSizeBound := 8
  vjp_cost_le op := by
    cases op <;> simp [ADable.vjp, subVjp, mulVjp, divVjp,
      ANFSeq.size_comp, ANFSeq.size, ANFFun.size]

/-- The cheap-gradient principle for the carrier-free reverse-mode transform.

If each primitive's reverse rule and arity are bounded by `B`, the transformed
graph contains at most `(B + 2)` instructions per primal instruction, plus terms
linear in the graph's input and output interfaces. -/
theorem ANFFun.vjp_size_le {Op S : Sig} [Op ⊆ S] [ZerosOp ⊆ S] [AddOp ⊆ S] [ADable Op
↪  S]
    [BoundedADable Op S] {Γin : Ctx} {outs : List Shape} (g : ANFFun Op Γin outs) :
    (ANFFun.vjp (S := S) g).size ≤
      (vjpSizeBound Op S + 2) * g.size + Γin.length + outs.length := by
  have h : ANFSeq.vjpCost S g.body ≤ vjpSizeBound Op S * g.size :=
    ANFSeq.vjpCost_le g.body
  rw [ANFFun.vjp_size, Nat.add_mul]
  omega

/-- The final result is `ANFFun.vjp_size_le' specialized to `BaurStrassenSig'. -/
theorem ANFFun.vjp_size_le_baurStrassenSig {Γin : Ctx} {outs : List Shape}
    (g : ANFFun BaurStrassenSig Γin outs) :
    (ANFFun.vjp (S := BaurStrassenSig) g).size ≤
      10 * g.size + Γin.length + outs.length := by
  simpa [vjpSizeBound] using ANFFun.vjp_size_le (S := BaurStrassenSig) g
```

A.5 Example Generations from our GPT-2-Style Model

We provide example generations from our pretrained and fine-tuned 125M-parameter GPT-2-style model, each with its complete prompt and response.

Generation 1

```
USER: What is the capital of France? Answer in one sentence.
ASSISTANT: The capital of France is Paris, a city renowned for its rich history,
↪  culture, and iconic landmarks.
```

Generation 2

```
USER: Write a haiku about the ocean.
ASSISTANT: Imagine a world where the ocean is a living, breathing entity that pulses
↪  with energy and life. The waves crash against the shore, their rhythmic pulse
↪  beating in harmony with the tides. The ocean's surface is a vast, barren expanse
↪  of sand, dotted with rocky outcroppings and coral reefs. The waves are a gentle,
↪  soothing melody that lulls the shore into a calm, peaceful state.

As the waves crash, the ocean's surface is a vast, barren expanse of sand, dotted
↪  with rocky outcroppings and coral reefs. The waves are a gentle, soothing melody
↪  that lulls the shore into a calm, peaceful state. The ocean's surface is a vast,
↪  barren expanse of sand, dotted with rocky outcroppings and coral reefs
```

Generation 3

```
USER: Explain what a hash table is to a beginner.
ASSISTANT: A hash table is a data structure that stores key-value pairs in a way that
↪  allows for efficient lookups and insertions. It's a way to store and retrieve
↪  data from a database, which is a collection of data that is stored in a table.

Think of a hash table as a way to store data that has been stored in a database. When
↪  you insert a key-value pair into a table, you're essentially creating a hash
↪  table that stores the key-value pairs. This hash table is called a hash table.

Here's how it works: when you insert a key-value pair into a table, you're creating a
↪  hash table that stores the key-value pairs. The hash table is called a hash
↪  table, and it's stored in a database.

Here's a basic example of a hash table
```

Generation 4

```
USER: List three prime numbers greater than 100.
ASSISTANT: Here are three prime numbers greater than 100:

1. 2
2. 3
3. 5
4. 7
5. 11
6. 13
7. 17
8. 19
9. 23
10. 29
11. 31
12. 37
13. 41
14. 43
13. 41
14. 43
15. 47
16. 53
17. 59
18. 61
19. 67
20. 71
21. 73
22. 79
23. 83
24. 89
25. 97
26. 97
27. 97
28. 97
29. 97
30. 97
```

A.6 Example Generations from Qwen 3.8 27B

We provide example prompts and responses from Qwen 3.8 27B.

Generation 1

```
USER: What is the capital of France? Answer in one sentence.
ASSISTANT: The capital of France is Paris.
```

Generation 2

```
USER: Write a haiku about the ocean.
ASSISTANT: White foam kisses sand, / Blue waves breathe in endless rhythm, / Salt air
↪  fills the lungs.
```

Generation 3

```
USER: Explain what a hash table is to a beginner.
ASSISTANT: A hash table is a way to store data so you can find it extremely fast.
↪  Think of it like a super-organized filing cabinet where you can jump straight to
↪  the right drawer without checking every single file.
```

Generation 4

```
USER: List three prime numbers greater than 100.
ASSISTANT: Here are three prime numbers greater than 100: 1. 101
```

A.7 Example Generations from Gemma 4 26-A4B

We provide example prompts and responses from Gemma 4 26-A4B.

Generation 1

```
USER: What is the capital of France? Answer in one sentence.
ASSISTANT: The capital of France is Paris.
```

Generation 2

```
USER: Write a haiku about the ocean.
ASSISTANT: The ocean is vast and deep, / A world of wonder and mystery, / Where life
↪  thrives and grows.
```

Generation 3

```
USER: Explain what a hash table is to a beginner.
ASSISTANT: A hash table is a data structure that stores key-value pairs. It uses a
↪  hash function to map keys to indices in an array, allowing for fast retrieval of
↪  values. The hash function takes a key as input and returns an index in the array
↪  where the corresponding value is stored. This allows for constant time lookup,
↪  insertion, and deletion of key-value pairs. However, hash tables can suffer from
↪  collisions, where two keys map to the same index. To handle collisions, hash
↪  tables use techniques such as chaining or open addressing.
```

Generation 4

```
USER: List three prime numbers greater than 100.
ASSISTANT: 101, 103, 107
```

A.8 Experimental Details

Every accelerator measurement below ran on a single NVIDIA B300 SXM6 (compute capability `sm_103a`, 275,040 MiB of HBM, driver 580.126.09) in float32. The language-model runs and the billion-parameter benchmarks used different OpenXLA builds, given in their respective sections.

Figure 4. Each AD system was run 32 times independently at even model depths 2-32. Benchmark processes were pinned to the one CPU core. No two benchmarks ran simultaneously. Plot shows median runtime. Empirical 5th and 95th percentiles are shown in the shaded regions.

A.8.1 Language Model Trained from Scratch

The model is nanochat's own depth-8 GPT (Karpathy, 2025), ported to DiffDog. It is architecturally identical to the reference: weightless RMSNorm, QK-norm, rotary embeddings, squared-ReLU feed-forward, sliding-window attention with value embeddings, untied embeddings and no biases.

Model and pretraining	
Layers, width	8, 512
Attention	4 heads of 128, no grouped-query; RoPE base 100,000; QK-norm
Feed-forward	2048, squared ReLU
Vocabulary, context	32,768; 2048 tokens
Parameters	125,829,354 total, 41,943,232 outside the embedding tables
Data	ClimbMix (Diao et al., 2025), 170 shards, no repeats
Tokens	503,316,480, twelve per non-embedding parameter
Step	1920 steps of 262,144 tokens (128 sequences of 2048)
Micro-batch	1, with gradient accumulation 128
Optimizer	Muon on the transformer matrices, AdamW on embeddings and scalars
Schedule	40-step warmup, flat to step 672, linear warmdown to 0.05 of peak
Precision	float32 throughout
Wall time	4,990 s (1 h 23 min) at 100,859 tok/s
Final validation	3.0731 nats/token, 0.9445 bits per byte

Three departures from the reference recipe are deliberate. We use a micro-batch of 1 with 128-fold gradient accumulation, so the optimizer sees the same 128-sequence mean gradient per step but the forward and backward passes run one sequence at a time; we train in float32 rather than bfloat16 with float8 linear layers; and we run on a single B300 rather than eight H100s. Everything else — token budget, batch size, optimizer groups, learning rates, betas, weight decays and schedule — follows nanochat's depth-8 configuration.

These timings are not comparable to nanochat's published speedrun, which trains a depth-24 model on eight H100s in bfloat16 and float8 for $187\times$ the training FLOPs. No depth-8 reference run exists to compare against. The runs also predate the benchmarks of Appendix A.8.2 and used an older OpenXLA build: PJRT API v0.98, CUDA 12.9 and cuDNN 9.13, against the CUDA 13.4 and cuDNN 9.26 of the benchmark toolchain.

Supervised fine-tuning.

Supervised fine-tuning	
Mixture	SmolTalk $\times 1$, MMLU `auxiliary_train` $\times 3$, GSM8K $\times 4$ (Allal et al., 2025; Hendrycks et al., 2021; Cobbe et al., 2021)
Data	789,759 conversations packed into 239,027 rows of 2049 tokens
Loss mask	assistant tokens only, 60.0% of the packed tokens
Steps	1868, one epoch, at the same 262,144 tokens per step
Schedule	no warmup; flat for the first half, linear to zero over the second
Wall time	6,120 s
Final validation	0.4178 bits per byte

Figure 3 plots per-task held-out perplexity from this run. Its MMLU curve should be read with care. Each MMLU target is two tokens, an answer letter and an end-of-turn marker, so most of the apparent improvement is the model learning to stop after one token rather than learning the answer. Measured directly on the letter alone, cross-entropy stays at $\ln 4$ from step 50 onward and ends at 1.4617, slightly worse than a uniform guess, and a probe of 1814 questions puts top-1 accuracy at 30.0% against a 25% chance baseline. The MMLU portion of the mixture, 38% of the conversations, bought almost nothing.

A.8.2 Billion-Parameter Model Benchmarks

Hardware and software. Every stack ran in float32 for both parameters and activations, at batch size 1, with PyTorch's matmul precision set to `highest` and TF32 disabled.

	DIFFDOG	JAX 0.10.2	PyTorch 2.9.1
CUDA	13.4	13.4	12.8 runtime
`ptxas`	13.4.92	13.4.92	12.9.86, via Triton
cuDNN	9.26.0	9.26.0	9.10.2
PJRT API	v0.112	v0.112	—
Lean	4.33.1	—	—
`transformers`	—	—	5.8.1
`peft`	—	—	0.21.0

DIFFDOG and JAX share one byte-identical OpenXLA PJRT plugin build and take their whole CUDA toolchain from the same `jax[cuda13]` wheels. PyTorch uses its own CUDA runtime and cuDNN, and its Triton bundled `ptxas` was replaced with CUDA 12.9's, which supports `sm_103a`.

Models.

	Qwen 3.8 27B	Gemma 4 26-A4B
Hugging Face repository	`Qwen/Qwen3.8-27B`	`google/gemma-4-26B-A4B`
Revision	`1d4bf0f2`	`24548b62`
Parameters	26,895,998,464	25,233,141,760
Feed-forward	Dense	Mixture-of-experts
Experts	—	128, routed top-8
Layers	64	30
Attention	Gated delta-net linear, full every 4th layer	25 sliding-window, 5 global
Attention heads	24 query, 4 key/value	—
Hidden size	5120	2816
Feed-forward width	17,408	704 expert, 2112 shared
Vocabulary	248,320	262,144

Measurement protocol. Each process performed one weight load, one cold compilation, which is not counted toward throughput, then discarded warm-up repetitions — one for the inference workloads, three for LoRA training — and then 10 timed repetitions. Each cell of Table 1 pools two independent processes: a *seed* process starting from empty autotuning and compilation caches, and a second process that reuses the caches the seed wrote. This gives $n = 20$ timed repetitions per cell, reported as the mean $\pm$ the sample standard deviation. No repetition was discarded or filtered. Timed processes held an exclusive lock on the GPU and an exclusive quiet-host lock on the CPU, so no two benchmarks ran simultaneously.

Workloads.

	Prefill	Decode	LoRA training
Prompt tokens	512	512	—
Tokens timed	512	127	1024 per step
Key-value cache slots	640	640	—
Throughput	$512/\text{wall}$	$127/\text{interval}$	$1024/\text{step}$
Timed repetitions	10	10	10 steps
Gemma MoE, DIFFDOG/JAX	Blocked sparse, $B_k = 64$	Routed-expert gather	Blocked sparse, $B_k = 256$
Gemma MoE, PyTorch	Per-expert loop	Batched matmul	Per-expert loop
`torch.compile` mode	`max-autotune-no-cudagraphs`	`max-autotune`	`max-autotune-no-cudagraphs`
PyTorch CUDA graphs	No	Yes	No

The prefill timer spans uploading the prompt to reading the first token back, with the language-model head applied only to the final row. For decode the prefill is run and synchronized before the timer

starts, so the reported figure is decode-only; the steps are then enqueued with asynchronous dispatch, each sampled token left resident on the device rather than read back per step. PyTorch captures the decode step into a CUDA graph; OpenXLA already issues each step as a single command buffer.

LoRA configuration.

General settings	
Form	$y = Wx + (\alpha/r)\, B(Ax)$, unmerged, base weights frozen
Rank r	16
α	16, giving scale $\alpha/r = 1$
Dropout, bias	0, none
Optimizer	AdamW, decoupled weight decay
Learning rate	10^{-3}, constant
β_1, β_2	0.9, 0.999
ϵ	10^{-8}
Weight decay	0.01, applied to the adapters only
Gemma adapted modules	115 modules, 230 adapter tensors, 11,489,280 trainable parameters
Qwen adapted modules	208 modules, 416 adapter tensors, 39,583,744 trainable parameters
Initialization	one shared deterministic init, byte-identical adapter values on all stacks

Data settings	
Dataset	`HuggingFaceH4/ultrachat_200k`, revision `8049631c`
Split	`train_sft`, 10,000-conversation subset, seed 42
Tokenizer	each model's own chat template
Gemma tokenizer	taken from the instruction-tuned (`-it`) release, revision `4d7ae498`
Loss mask	assistant tokens only, end-of-turn included (≈77% of tokens for Gemma, ≈74% for Qwen)
Packing	stream concatenated and chunked into 1025-token pieces (1024 inputs and shifted targets); partial tail dropped
Objective	masked next-token cross-entropy, averaged over the masked tokens
Row schedule	row 0 compiles, rows 1–3 warm up untimed, rows 4–13 are timed

The chat template that formats conversations into turns ships with Gemma's instruction-tuned release rather than the base checkpoint, so that is where the tokenizer comes from; the weights being benchmarked are still the base checkpoint of the model table above. These rows time one complete training step — forward, backward with respect to the adapters, and the AdamW update — rather than a run to convergence. The adapters target:

- **Gemma**: the attention query, key, value and output projections, dropping the value adapter on the global-attention layers, where key and value share a projection.
- **Qwen**: the query, key, value and output projections of the full-attention layers, and the fused QKV input, z input and output projections of the gated delta-net layers.

PyTorch runs the step under `torch.compile` with a compiled optimizer step; JAX runs one ahead-of-time compiled jitted step with the adapters and optimizer state donated; DIFFDOG runs one compiled XLA executable per step.

Correctness. On decode, all three stacks emit identical greedy token identifiers: the same 128 tokens on every repetition of both models and in both processes. Prefill produces only one token, which likewise agrees. The LoRA rows measure throughput rather than output, so there the equivalent check is that every stack performs the same work per step: the same data rows in the same order, the same adapted modules, the same adapter initialization, and the same hyperparameters and shapes, with the mixture-of-experts layer routed sparsely top-8 in all three. For reference, the Qwen training losses agree across stacks to roughly 10^{-5} relative, while the Gemma losses are not reproducible to that precision even between two processes of the same stack.

Table 2: Full throughput measurements for open-weight models, in tokens per second, as mean ± sample standard deviation over $n = 20$ timed repetitions across two processes, except the stock official library ($n = 10$, one process). Decode excludes prefill except where marked. The Gemma JAX baseline is DeepMind's `gemma` library 4.0.1 on our toolchain, patched only so that it performs the same work as the other stacks. †Those patches include replacing its dense `ragged_dot` mixture-of-experts path with the same top-8 grouped matmul DIFFDOG uses, so roughly 70% of its prefill and fine-tuning matrix-multiply work runs in our kernel and that column largely measures the library's non-mixture-of-experts code. The Qwen JAX baseline, and ‡the supplementary Gemma row, are instead written from scratch. The stock row is the library unmodified apart from float32.

Model	System	LoRA (T=1024)	Prefill (T=512)	Decode
Qwen 3.8 27B	DIFFDOG (ours)	520.58 ± 1.57	1103.19 ± 4.72	41.34 ± 0.63
	JAX, written from scratch	512.67 ± 0.52	1097.23 ± 4.24	41.57 ± 0.64
	PyTorch (HF)	452.56 ± 28.40	937.84 ± 0.71	50.37 ± 0.05
Gemma 4 26-A4B	DIFFDOG (ours)	1356.36 ± 3.68	3424.82 ± 13.20	139.03 ± 4.69
	DeepMind `gemma`, patched†	1342.64 ± 3.47	2230.72 ± 31.05	145.98 ± 0.57
	PyTorch (HF)	1180.72 ± 31.76	2077.04 ± 4.54	137.28 ± 2.69
	JAX, written from scratch‡	1353.76 ± 4.79	3431.33 ± 14.39	132.52 ± 0.08
	DeepMind `gemma`, stock	83.92 ± 0.04	157.96 ± 0.14	34.52 ± 0.02*

*Not comparable to the other decode figures. The official library's sampler is a single fused jitted loop, so decode cannot be separated from prefill and this figure is end-to-end, dividing the generated tokens by the prefill *and* decode time together. It also allocates a 4096-slot key-value cache rather than 640.

Table 2 reports the throughput of Table 1 at full precision, together with the official DeepMind Gemma library discussed below.

Per-process results and sources of variation. Per-process means are released with the benchmark harnesses rather than reproduced here. The standard deviations are not pure run-to-run noise. On several cells the two processes differ systematically, because autotuning picks different kernels from a cold cache than from a warm one — DIFFDOG's Gemma decode runs at 134.46 against 143.60 tok/s — and two unfiltered PyTorch first-repetition effects inflate them further: the first timed Qwen LoRA step of the cached process reaches 330.3 tok/s against roughly 459 for the rest, and CUDA-graph recording spills into the first Gemma decode repetition. With only two processes per cell we attach no confidence intervals, and caution against reading significance into the small DIFFDOG/JAX gaps.

Why PyTorch trails on Gemma prefill and LoRA training. Hugging Face's fast grouped mixture-of-experts kernel, `torch._grouped_mm`, accepts only bfloat16, so in float32 the model falls back to `grouped_mm_fallback`: a per-expert `torch.mm` loop with a device-to-host read per layer. That read, `aten._local_scalar_dense`, is triggered by `offs.tolist()` and cannot be compiled by TorchInductor, so every mixture-of-experts layer graph-breaks and runs eagerly. Under plain `max-autotune` the resulting CUDA graphs then crash the fine-tuning backward pass with "accessing tensor output of CUDAGraphs that has been overwritten", which is why the prefill and LoRA configurations use `max-autotune-no-cudagraphs`.

Our JAX ports. The Qwen JAX baseline is written from scratch, because Qwen 3.8 27B has no official JAX implementation. It follows the Hugging Face reference semantics, with one exception: the gated delta-net chunk solve uses DIFFDOG's triangular-inverse formulation, chosen by a pre-registered timing test in which `lax.linalg.triangular_solve` came out 0.5% slower. The Gemma JAX baseline is the official library, patched as described below. The supplementary Gemma translation in Table 2 is a separately written JAX port of DIFFDOG's own model, blocked-sparse mixture-of-experts algorithm included, so it is not an independent implementation either. No JAX column shares source code or emitted kernels with DIFFDOG, but every column in both tables

compiles through the same XLA build. Where the two agree, then, two separately written programs reached similar speed through one compiler; that is weaker than two independent systems agreeing. The decode agreement reported above establishes that all three stacks implement the same model, not that they agree bitwise: greedy decoding is insensitive to differences too small to change an argmax, and the Gemma training losses are not reproducible even between two runs of the same stack.

The Gemma JAX baseline. Gemma's JAX column is DeepMind's `gemma` library 4.0.1 on our toolchain, patched only where the stock library would otherwise do different work from the other stacks. Run stock it is an order of magnitude slower, the last row of Table 2, because on GPU it preserves top-8 routing but not its sparsity: OpenXLA's default lowering of `jax.lax.ragged_dot` pads each token's row into a dense tensor and contracts it against all 128 experts, masking the 120 unrouted products to zero. Results stay correct; the cost is roughly $16\times$ the routed work. This is OpenXLA's documented fallback, which its developers describe as producing "numerous unnecessary calculations" (OpenXLA contributors, 2026; OpenXLA, 2025); the grouped Triton kernel meant to replace it had not shipped, and the opt-in Pallas path is broken in released wheels (JAX contributors, 2026). It is also why neither our translation nor DIFFDOG uses `ragged_dot`: the limitation is widely reported (AMD ROCm, 2026; dew contributors, 2026), and production mixture-of-experts stacks ship their own grouped-matmul kernels instead (MaxText contributors, 2026; NVIDIA, 2026; LMSYS Org, 2025; rdyro, 2026). The patches replace that path with the same top-8 grouped matmul DIFFDOG uses, adapt the router to read the post-attention residual as the Hugging Face reference does rather than a normalized input, and supply LoRA target modules, which the stock `gm.nn.LoRA` does not provide for Gemma 4. The router change is what brings the library's output into agreement: unmodified it is the one configuration in Table 2 whose greedy tokens differ from the others, and with the change $30{,}719$ of $30{,}720$ routed expert sets match the reference, the exception being a float32 near-tie. The library's routing, dispatch, combine, attention, cache and optimizer code is left as it ships. Because the grouped matmul is ours, roughly 70% of the prefill and fine-tuning matrix-multiply work in that column runs in our kernel, so it is best read as a measurement of the library's remaining code rather than of JAX against DIFFDOG. Every patched result is gated on emitting the same 128 decode token identifiers as the other stacks and on executing sparse top-8 mixture-of-experts HLO with expert work equal to ours. Profiling with Nsight Systems attributes the library's decode lead to less GPU work per step, 6.75 against 6.93 ms, not to dispatch: DIFFDOG already runs each decode step as a single CUDA graph and idles 0.1% of the time between steps, while the library's jitted loop reads its predicate on the host every step. The difference is in how a few operations are formulated, chiefly that DIFFDOG computes the key/value and gate/up projections as separate products and materializes their intermediates where the library uses one stacked reduction.

Artifacts. We release the benchmark harnesses, the per-process result files and one-command reproduction scripts for every number reported here at `https://github.com/REDACTED/REDACTED`, on the `final-repro` branch. Four commands cover the paper: `setup.sh`, `download_weights.sh`, `run_cell.sh` and `make_tables.sh`, the last of which regenerates the reported tables. `repro/DATA_MAP.md` maps each item in the paper to the command that produces it. The one exception is the TorchLean estimate of footnote 4, which is an extrapolation rather than a measurement of our own system.