

Verified Scientific Machine Learning in Lean

Robert Joseph George

The logo for Caltech, featuring the word "Caltech" in a bold, orange, sans-serif font.

Formal mathematics

Formalizing Fermat's Last Theorem

Sep 4, 2026



Anthropic · Fermat's Last Theorem

Learning more about Claude's mathematical capabilities

Aug 10, 2026



Anthropic · Zeros of the zeta function

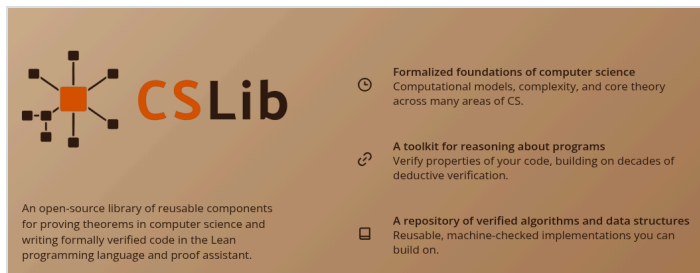
Finite time blowup for Navier–Stokes and Euler equations

This repository contains Lean 4 formalizations of the results presented in "[Finite time blowup for Navier–Stokes](#)" and "[Finite time blowup for the Euler equation](#)" by OpenAI.

OpenAI · Forced Navier–Stokes; unforced Euler

- **Mathlib:** a library of formal mathematics.
- **Tau Ceti:** AI formalization guided by human-written roadmaps.
- **Lean Pool:** an archive of independent formalizations.
- **MAP:** Jared Duker Lichtman's proposal for a formal database of mathematics.

Beyond pure mathematics



- **CSLib:** computer science.
- **Aeneas:** program verification (Rust).
- **Physlib:** physics.
- **Chemistry:** adsorption and gas laws.
- **Econlib:** economic and political theory.
- **Statlib:** mathematical statistics.

And many more I can't cover here.

Econlib

A Lean 4 library for economic and political theory.

Econlib formalizes economic and political theory in Lean 4 — preferences, probability, optimization, game theory, mechanism design, social choice, general equilibrium, and the supporting mathematics — with an API designed to help you formalize your research.

ABOUT ECONLIB

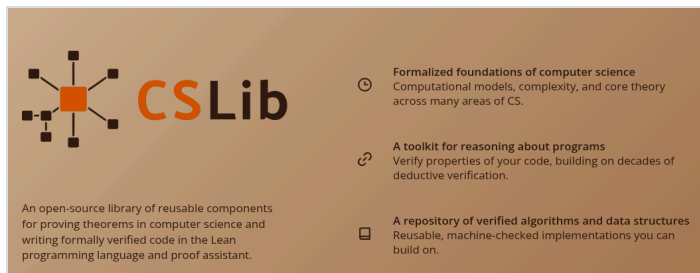
The paper

An introduction to Econlib's design principles, scope, key results, and future plans.

GitHub source

The full library source and issue tracker.

Beyond pure mathematics



- **CSLib:** computer science.
- **Aeneas:** program verification (Rust).
- **Physlib:** physics.
- **Chemistry:** adsorption and gas laws.
- **Econlib:** economic and political theory.
- **Statlib:** mathematical statistics.

And many more I can't cover here.

Econlib

A Lean 4 library for economic and political theory.

Econlib formalizes economic and political theory in Lean 4 — preferences, probability, optimization, game theory, mechanism design, social choice, general equilibrium, and the supporting mathematics — with an API designed to help you formalize your research.

ABOUT ECONLIB

The paper An introduction to Econlib's design principles, scope, key results, and future plans.	GitHub source The full library source and issue tracker.
---	--

Applied mathematics also needs
numerical libraries and machine learning in Lean.

What is Lean?

```
-- An executable definition
def relu (x : Int) : Int := max 0 x

#eval relu (-3)      -- 0

-- A theorem about relu
theorem relu_nonneg (x : Int) :
  0 ≤ relu x := by
  simp [relu]
```

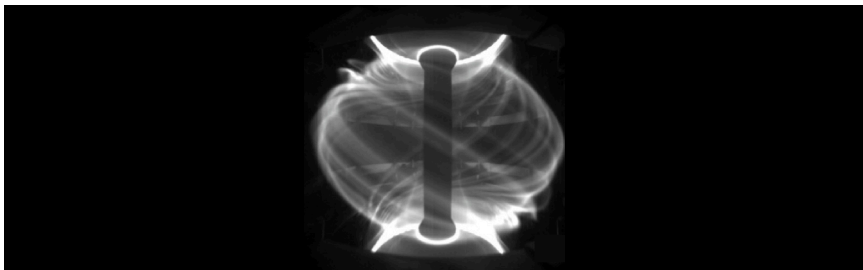
- **An interactive theorem prover**
Proofs checked by Lean's kernel.
- **A functional programming language**
Programs compiled to native code.

Programs and theorems refer to the same definitions.

Plan

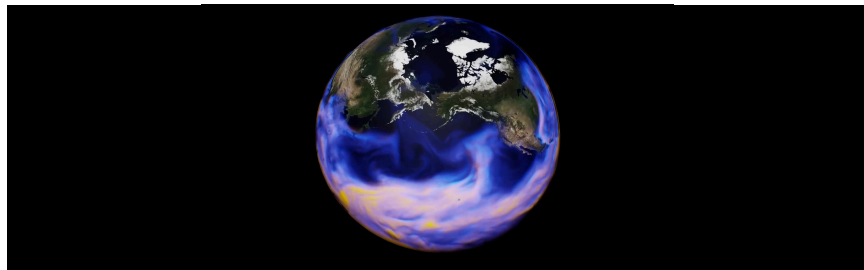
- | | | |
|----|--------------------|--|
| 1. | TorchLean | Models, training, and verification |
| 2. | FloatLib | Floating-point arithmetic and error bounds |
| 3. | Future work | Connecting to formal PDE analysis |

AI for science at Tensor Lab



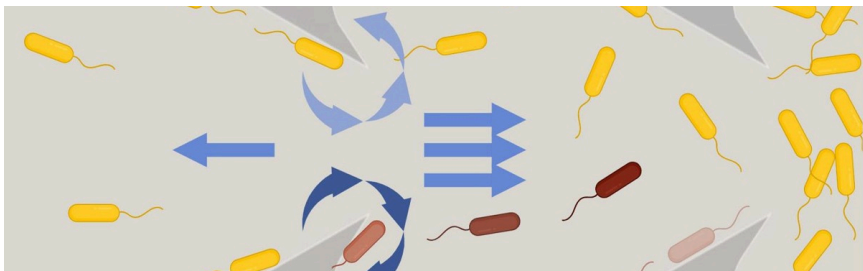
Nuclear fusion

Plasma forecasting



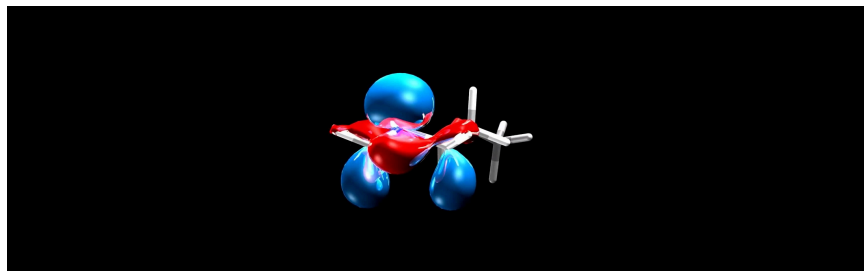
Weather forecasting

FourCastNet



Medical devices

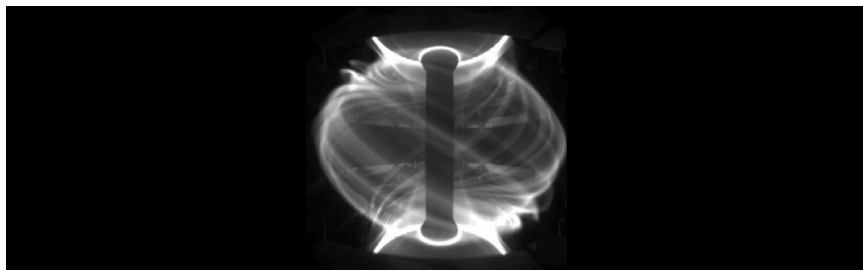
Design optimization



Molecular design

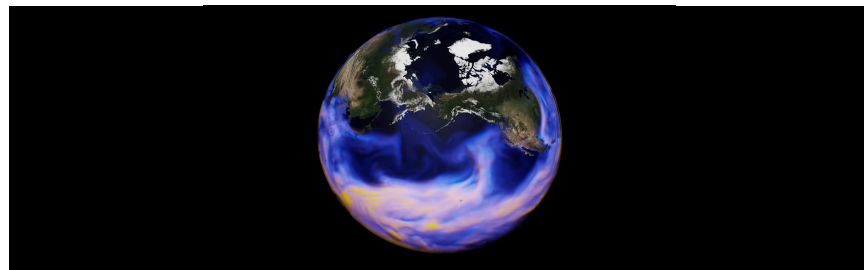
Quantum chemistry

AI for science at Tensor Lab



Nuclear fusion

Plasma forecasting



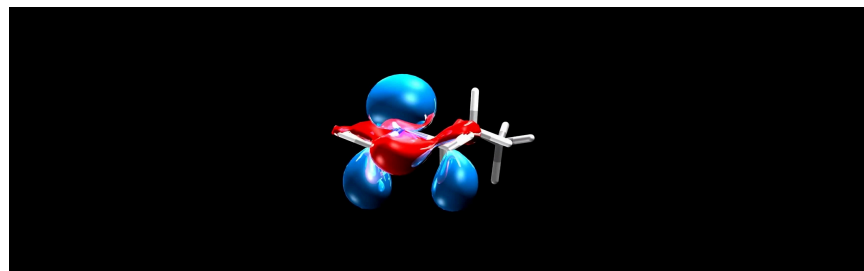
Weather forecasting

FourCastNet



Medical devices

Design optimization



Molecular design

Quantum chemistry

These applications need foundations for verified ML and numerics.



TorchLean: Formalizing Neural Networks in Lean

TORCHLEAN: FORMALIZING NEURAL NETWORKS IN LEAN

A PREPRINT

Robert Joseph George¹, Jennifer Cruden², Will Adkins³,

Xinyu Zhang¹, Huan Zhang¹, Anima Anandkumar¹

¹California Institute of Technology

²Washington University in St. Louis

³University of Illinois Urbana-Champaign

ABSTRACT

Neural networks are increasingly deployed in scientific, safety critical, and mission critical pipelines, yet verification and analysis are often performed outside the programming environment that defines, runs, and exports the model. This separation creates a semantic gap between the executed network and the analyzed artifact: guarantees can depend on implicit conventions about operator semantics, tensor layouts, preprocessing, floating-point behavior, graph transformations, accelerated kernels, and externally produced certificates. We present TorchLean, a unified framework for formalizing, executing, and verifying neural networks in Lean 4. TorchLean transforms learned models as executable programs and first-class mathematical objects with a shared semantics for computation, verification, and theorem proving. The framework provides a PyTorch-style API for typed tensors, layers, objectives, optimizers, automatic differentiation, and graph programs, together with eager and compiled execution paths that lower to a common computational graph representation. TorchLean supports exact and finite-precision tensor semantics, verified reverse-mode differentiation for supported graph programs, interval and affine bound propagation, CROWNLRPA style certificate checking, import and export workflows, and CUDA-backed execution through explicit FFI boundaries. It also includes formal semantic layers for modern learning systems, including attention and FlashAttention, state-space sequence models, diffusion and sampling processes, probability kernels, reinforcement learning objectives and Markov decision processes, and self-supervised objectives such as masked autoencoding, BERT-style predictive views, and variational-based anti-collapse losses. Together, these components provide a semantic foundation for verified machine learning, where executable neural network artifacts, verification procedures, runtime boundaries, and mathematical claims can be stated and related inside a single theorem-proving environment¹.

Keywords neural networks · verification · formalization · automatic differentiation · Lean

1 Introduction

Neural networks are increasingly embedded in systems where mistakes have physical or societal cost, and the machine learning community is correspondingly placing more weight on guarantees that go beyond empirical accuracy: robustness to perturbations, satisfaction of hard safety and stability constraints, and conservative bounds on quantities derived from a network's computation. This has driven an active verification ecosystem spanning *ad-hoc* approaches that reduce verification queries to constraint solving, such as Reluplex and Marabou for piecewise-linear networks (Katz et al., 2017, 2019), as well as scalable relaxation and abstract interpretation methods that compute certified enclosures by bound propagation Gowal et al. (2018), Zhang et al. (2018), Singh et al. (2019). Surveys and system papers emphasize that such techniques are central to validating learning-enabled components in autonomy, control, and other high-assurance pipelines Xiang et al. (2018). Despite this progress, turning verifier outputs into dependable guarantees remains difficult in practice. The issue is not only the strength of a verifier, but also the meaning of the artifact being verified. A trained network may appear as a PyTorch nn.Module, an exported ONNX or FX graph, a compiler IR, a

¹Code is available at <https://github.com/lean-dojo/TorchLean>

A Lean library for neural networks.

- Shape-indexed tensors with a PyTorch-like API.
- Autograd and training on CPU/CUDA.
- Compiler and derivative theorems; numerical error bounds.

George et al. · arXiv:2602.22631

github.com/lean-dojo/TorchLean

Why TorchLean?

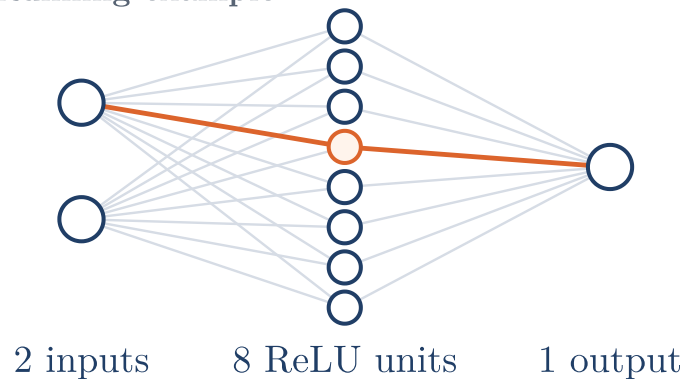


Why TorchLean?



Verification must account for export, compilation, and execution.

Running example



A two-layer network.



PyTorch and TorchLean

PyTorch

```
model = nn.Sequential(  
    nn.Linear(2, 8),  
    nn.ReLU(),  
    nn.Linear(8, 1),  
)
```

TorchLean

```
def model :=  
  nn.Sequential![  
    nn.linear 2 8,  
    nn.relu,  
    nn.linear 8 1  
  ]
```

Tensor shapes are part of the Lean model type.

Tensor representation

Tensor $\alpha [d_1, \dots, d_k]$

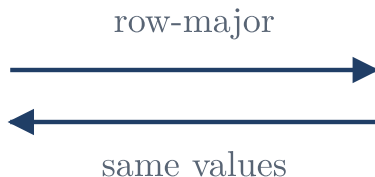
```
structure Rep ( $\alpha$  : Type u) (s : Shape)
  [storage : Storage  $\alpha$ ] where
  buffer : storage.Buffer
  size_eq : storage.size buffer = Shape.size s
```

- One contiguous, row-major buffer.
- Length certified by the shape.
- The proof is erased at runtime.

Coordinate view

1	2	3
4	5	6

$(\text{Fin } 2 \times \text{Fin } 3) \rightarrow \alpha$



Contiguous buffer

1	2	3	4	5	6
---	---	---	---	---	---

length = 2×3

Tensors and Mathlib

Matrix algebra

$$\text{Tensor } \mathbb{R} [m, n] \simeq \text{Matrix } (\text{Fin } m) (\text{Fin } n) \mathbb{R}$$

Proved correspondence for transpose, multiplication, and trace.

Analysis

$$\text{Tensor } \mathbb{R} s \simeq_{\mathbb{R}} \text{EuclideanSpace } \mathbb{R} (\text{Coord } s)$$

Norms, inner products, and Fréchet derivatives on real tensors.

Mathlib's triangle inequality applies directly:

$$\forall x, y : \text{Tensor } \mathbb{R} s, \quad \|x + y\| \leq \|x\| + \|y\|.$$

Shape checking

```
def broken :=  
  nn.Sequential! [  
    nn.linear 2 8,  
    nn.relu,  
    nn.linear 7 1  
  ]
```

Shape checking

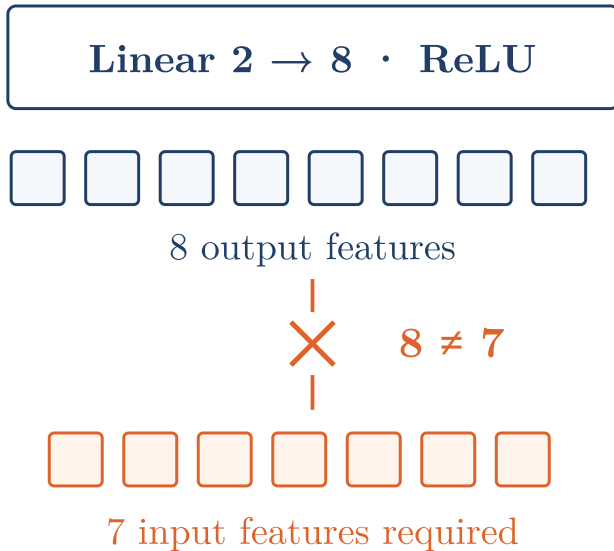
```
def broken :=  
  nn.Sequential! [  
    nn.linear 2 8,  
    nn.relu,  
    nn.linear 7 1  
  ]
```

Matching dimensions

`nn.linear 8 1`

Lean rejects the composition before execution.

The connection cannot be typed.



Training

```
let trainer := Trainer.new model
  { objective := .meanSquaredError
    optimizer :=
      optim.sgd { learningRate := 0.05 }
    execution := .typedGraph
    device := .cpu
    arithmetic := .ieee }
let trained ← trainer.train data
  { steps := 200, samplesPerStep := 16 }
trained.printSummary
```

Configuration

Choose the loss, optimizer, device, and arithmetic.

Models

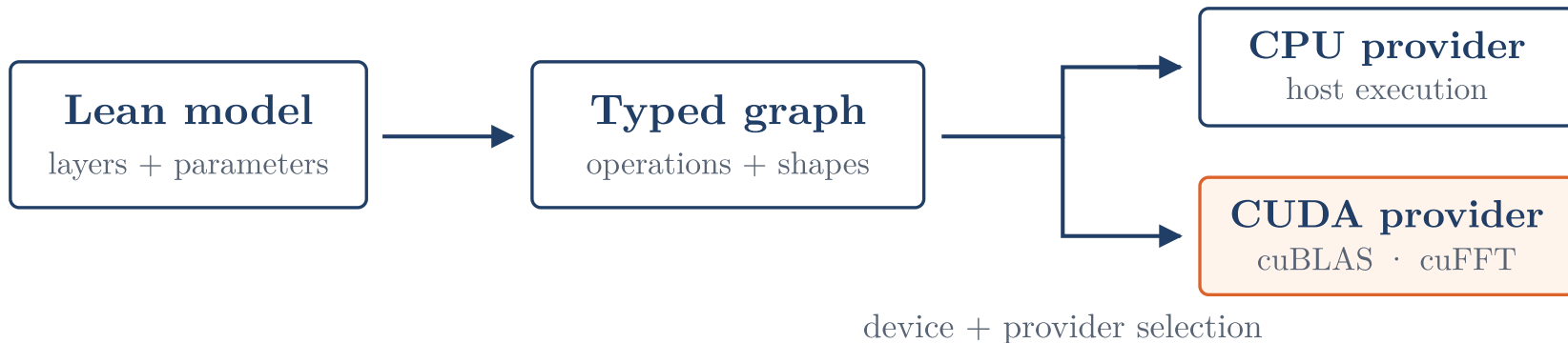
MLPs, Transformers, and neural operators.

Theorems

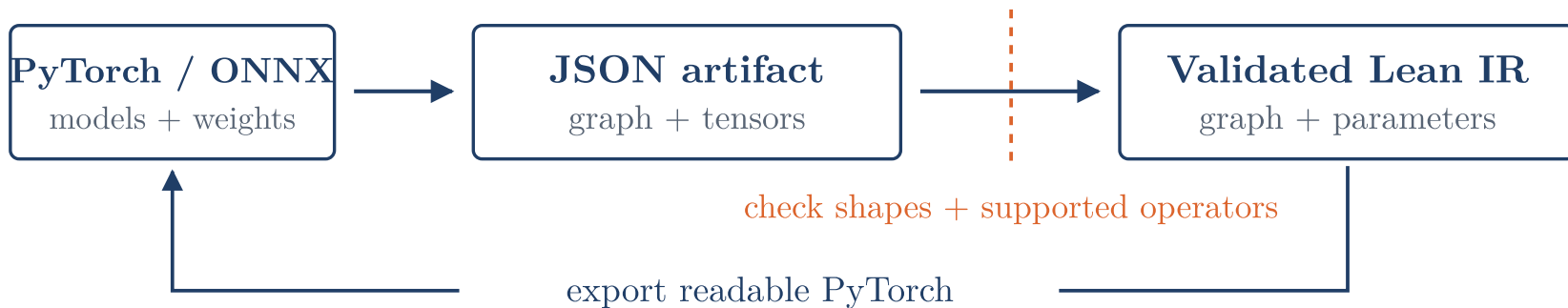
Approximation, robustness, and learning stability.

Backends and interoperability

EXECUTION



INTEROPERABILITY



Autograd correctness

Typed graph IR

$$f : X \rightarrow Y$$

↓ lowering

Exact-real tape

Saved values and reverse rules

↓

Reverse pass

Seed with output cotangent $\bar{y}$

For the graph's real forward map $f : X \rightarrow Y$, assume correct local derivative rules.

Adjoint Fréchet derivative

$$\forall x \in X, \forall \bar{y} \in Y,$$

$$\text{backward}_{\mathbb{R}}(x, \bar{y}) = Df(x)^* \bar{y}.$$

Global: derivative hypotheses at every input.

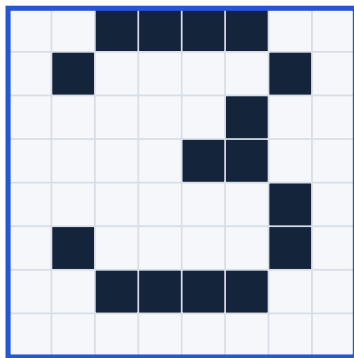
Pointwise: hypotheses at the values reached by this input.

ML theory

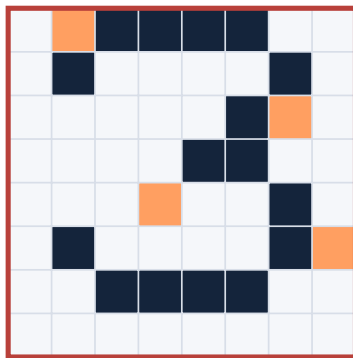
Approximation	Universal approximation theorem for neural networks.
Optimization	Gradient descent converges for smooth, strongly convex objectives.
Learning theory	Replace-one stability for bounded scalar ridge regression.
Reinforcement learning	Bellman contraction, fixed-point uniqueness, and value-iteration error bounds.

Further results on Hopfield dynamics, diffusion, and quantization.

Certified robustness



original image



any allowed perturbation

Every pixel may change within the radius.

Fix image $x \in \mathbb{R}^{64}$, label y , and radius $\varepsilon \geq 0$.

$$\forall \delta \in \mathbb{R}^{64}, \quad \|\delta\|_{\infty} \leq \varepsilon \implies \\ \arg \max_k f_k(x + \delta) = y.$$

Sound logit bounds $\rightarrow$ positive margin

$$\underline{f}_y > \max_{k \neq y} \bar{f}_k$$

IBP · interval bounds

CROWN / α, β -**CROWN** · affine bounds

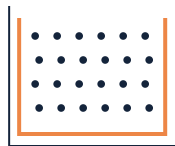
Lean · certificate replay

Scientific ML

Models and training in TorchLean.

Physics-informed neural networks

$$(x, t) \mapsto u_{\theta}(x, t)$$



- Tanh neural field.
- Sampled PDE residual and initial/boundary losses.

Fourier neural operators

$$\mathcal{G}_{\theta} : u_0 \mapsto \hat{u}(\cdot, T)$$



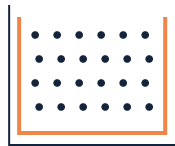
- MSE loss and Adam.
- Portable DFT or CUDA/cuFFT models.

Scientific ML

Models and training in TorchLean.

Physics-informed neural networks

$$(x, t) \mapsto u_{\theta}(x, t)$$



- Tanh neural field.
- Sampled PDE residual and initial/boundary losses.

Fourier neural operators

$$\mathcal{G}_{\theta} : u_0 \mapsto \hat{u}(\cdot, T)$$



- MSE loss and Adam.
- Portable DFT or CUDA/cuFFT models.

For PINN certification: bound the residual between samples.

GPT-2-scale training

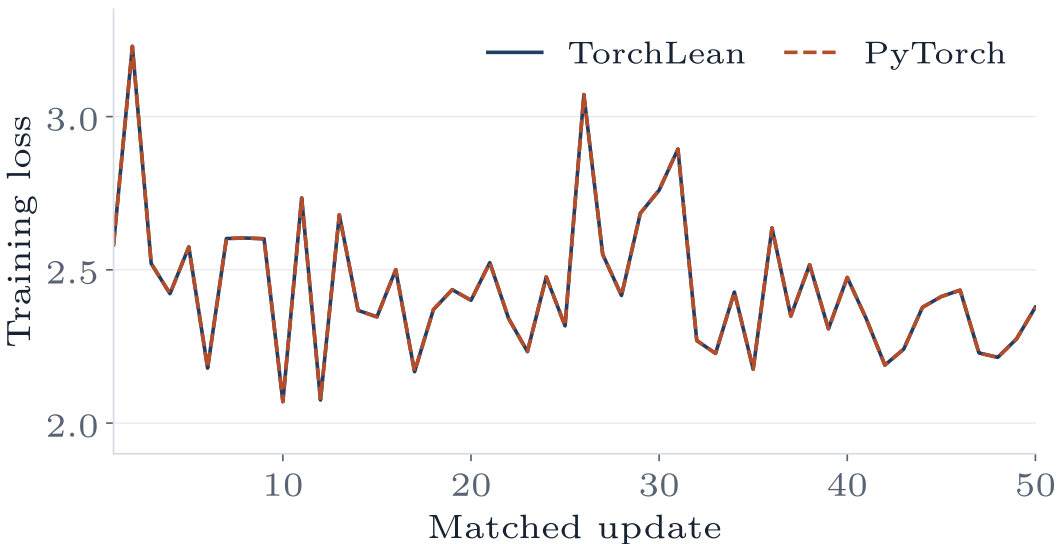
12 blocks · 12 attention heads · width 768 · context 1,024

124.4 million

parameters

2.319 billion

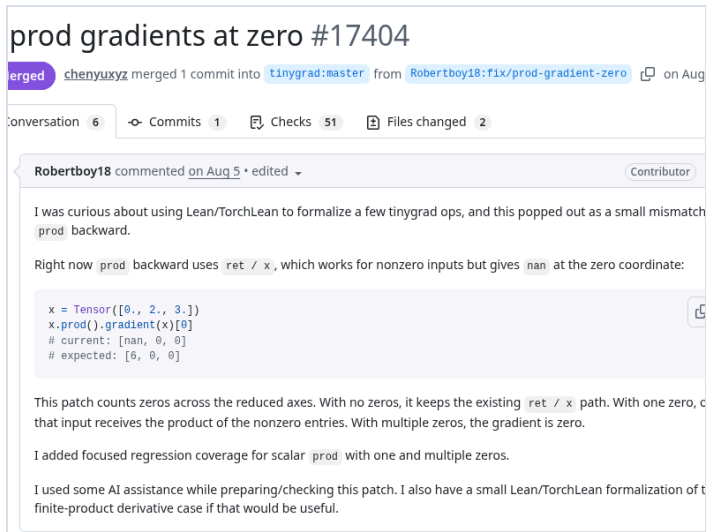
training tokens · one A100



50 matched updates: same checkpoint, batches, optimizer.

Median throughput: PyTorch $3.43\times$ TorchLean.

Fixes across libraries



tinygrad #17404 · product gradients at zero

tinygrad

Product derivatives at zero.

Hugging Face PEFT

Atomic safe merges for LoRA and IA3.

AWS Neuron nkiPy

Slicing and indexing.

PyTorch

Attention and loss functions.

NASA pvslib (PVS)

Floating-point wrappers and error bounds.

Thinking Machines Lab

Floating-point bug.

+ more libraries

Models

- MLPs, CNNs, ResNets, vision Transformers.
- RNNs, LSTMs, GPT-2, Mamba.
- Diffusion, autoencoders, VAEs, GANs, FNOs, KANs.
- DQN and PPO.
- Linear, logistic, ridge regression; SVMs.
- Trees, forests, boosting, naive Bayes, k-NN.
- PCA, GMMs, HMMs, Hopfield networks.
- GNNs, S4, Seq2Seq, U-Net, VQ-VAE.

Shared definitions for execution and proofs

Shapes, graph semantics, derivatives, approximation, optimization, floating-point error, robustness, and certificate checking.

Verification evidence

Evidence	Example	What it establishes
Lean theorem	Graph lowering; reverse-mode AD	A mathematical conclusion under stated hypotheses.
Executable checker	Graph validation; certificate replay	The artifact satisfies the checked predicate.
Runtime test	Backend parity; training runs	Observed behavior on tested executions.
Conditional theorem	Native Float32 agreement	A conclusion assuming runtime agreement.
External boundary	CUDA, LibTorch, Python exporters	Delegated behavior remains trusted.

FloatLib

An arbitrary-precision floating-point library in Lean.

 Verified floating point in Lean
Search

Overview
CHAPTERS
01 Getting started with FloatLib
02 Representing real numbers in binary
03 A short history of floating point
04 Proving floating-point arithmetic correct
05 Executable arithmetic and its proofs
06 The numerical models
07 The mathematics of rounding
08 IEEE binary and decimal formats
09 Low-precision formats for machine learning
10 P3109 formats
11 Posits and the quire
12 Fixed point, logarithmic numbers, and quantization
13 Kernels: arithmetic on words and limb arrays
14 Choosing an arithmetic backend
15 Performance and native floats
16 Related work and external validation
SOURCES
Dependency graph
Reference list

FloatLib is a Lean 4 library for executable floating point arithmetic with machine-checked proofs of its numerical behavior.

Robert Joseph George
Caltech

Will Adkisson
Washington University in St. Louis

Anima Anandkumar
Caltech

We built FloatLib so that the floating point code we run is also the code we can prove things about. The $x + y$ in a program is the same operation covered by the theorem, down to rounding and the meaning of the result bits.

A few highlights we're excited about:

- **Your formats and rounding rules.** You can [define your own format](#) within the same framework. For binary layouts, choose the exponent and fraction widths, bias, and encoding policy for zeros, infinities, and NaNs. For a new representation or rounding rule, our shared interfaces connect your executable operations to their numerical specification through Lean proofs.
- **The classical results, connected to code.** Correct rounding, half-up error bounds, and Sterbenz's lemma for exact subtraction are part of the [rounding theory](#). We can use them to reason about the arithmetic in a program.
- **NaNs, infinities, and both zeros have a place in the proofs.** The [IEEE model](#) keeps their encodings and behavior explicit, including comparisons and exception flags. A theorem about the complete result word can distinguish things that equality over the reals cannot.
- **Many formats, shared foundations.** IEEE binary and decimal, [small ML formats](#), [posits](#), and [P3109](#) sit alongside fixed point, logarithmic, codebook, and block-scaled representations. Posits also have exact quire accumulation within its capacity and real-rounding proofs for roots, powers, exponentials, and logarithms.
- **Speed is a major part of the work.** We have put substantial effort into [backend kernels](#): small lookup tables, machine-word algorithms, and limb arithmetic for wider values. The planner selects an implementation for each operation, and every certified choice proves agreement with the same specification. Lean erases proof terms when compiling, so arithmetic does not check its theorem again at runtime [31]. Our goal is to approach MPFR's speed while keeping that connection to the proofs. The [performance chapter](#) shows the measurements from 2 to 4,096 bits and where the gap remains.
- **Independent implementations give us another useful check.** FloatLib matched Berkeley TestFloat on more than **102 million cases** for the IEEE formats and operations we checked. We also compare with MPFR, posit libraries, and published small-format tables. There are some subtle differences with SoftPosit, which we [explain in the posits chapter](#); the [external validation chapter](#) gives the inputs and results.



■ Formats

Binary, decimal, posits, FP8, FP6, and FP4.

■ Correctness

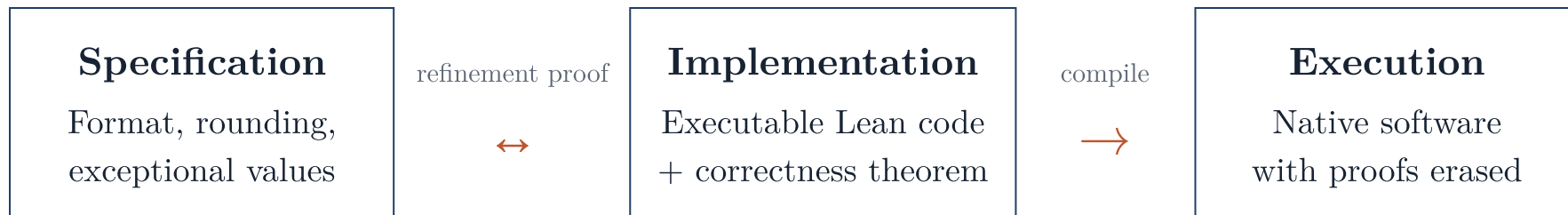
Certified operations match their floating-point specification.

■ Error bounds*

Rounding analysis and interval enclosures.

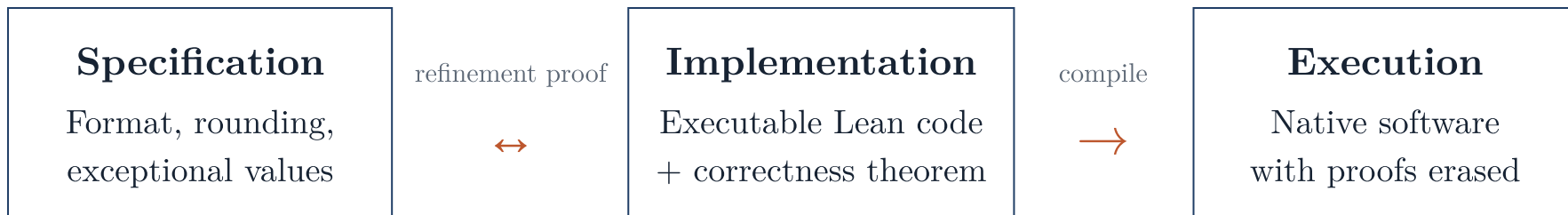
arxiv.org/abs/2609.19352

Specifications and backends



- Generic arithmetic provides the reference implementation and fallback.
- Specialized backends use tables, word kernels, or limb arrays.
- The format, operation, and storage determine the backend.

Specifications and backends

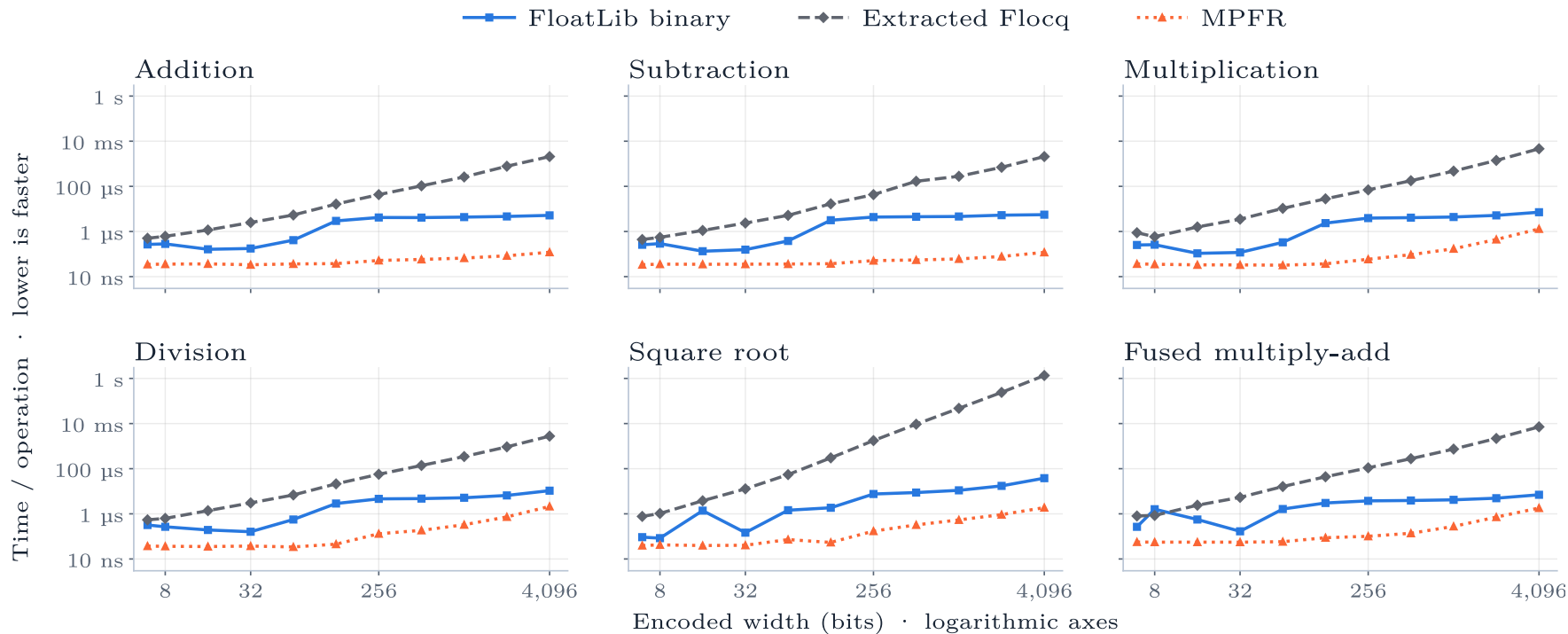


$$\forall x, y, \quad \text{add}_{\text{exec}}(x, y) = \text{add}_{\text{spec}}(x, y)^*$$

- Generic arithmetic provides the reference implementation and fallback.
- Specialized backends use tables, word kernels, or limb arrays.
- The format, operation, and storage determine the backend.

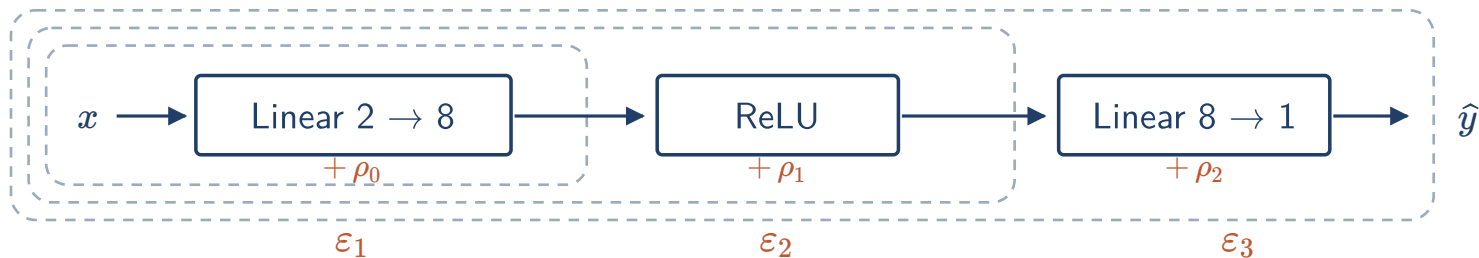
Formats and performance

Matched binary formats · Shared fixtures and rounding · 6–4,096 bits



Composing error bounds

Each layer propagates input error and adds rounding error.

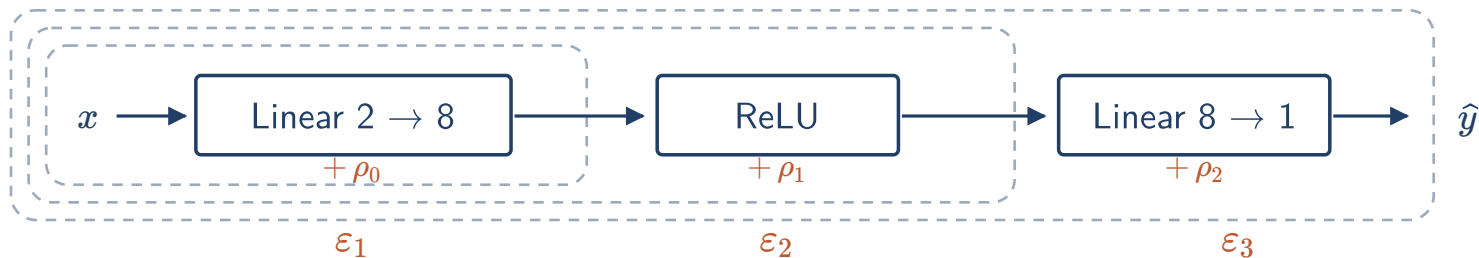


$$\underbrace{\varepsilon_{k+1}}_{\text{next error}} \leq \underbrace{L_k \varepsilon_k}_{\text{propagated error}} + \underbrace{\rho_k}_{\text{local rounding}}$$

Requires: a Lipschitz bound on each layer's input range, valid local rounding bounds, and finite outputs.

Composing error bounds

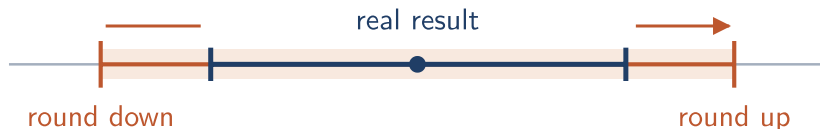
Each layer propagates input error and adds rounding error.



$$\underbrace{\varepsilon_{k+1}}_{\text{next error}} \leq \underbrace{L_k \varepsilon_k}_{\text{propagated error}} + \underbrace{\rho_k}_{\text{local rounding}}$$

Outward rounding

Round the lower endpoint down and the upper endpoint up.



Requires: a Lipschitz bound on each layer's input range, valid local rounding bounds, and finite outputs.

My hope for the future

Toward a mathlib for machine learning and scientific computing.

- Formalize models and algorithms from ML papers.
- Train and evaluate them in Lean.
- Prove their properties and bound numerical error.
- Find and fix bugs in ML infrastructure.
- Connect to the growing range of libraries across Lean and beyond.

Acknowledgments

My advisor

Anima Anandkumar

Collaborators

Jennifer Cruden

Xiangru Zhong

Peicong Song

Will Adkisson

Huan Zhang

All contributors to TorchLean and FloatLib.

The Lean and Mathlib communities.

Funding: DARPA Exponentiating Mathematics (expMath).

Thank you to IPAM, UCLA, and the workshop organizers.

Thank you:)

github.com/lean-dojo/TorchLean

github.com/lean-dojo/FloatLib

Selected references

George, Cruden, Adkisson, Zhong, Zhang & Anandkumar.

TorchLean: Formalizing Neural Networks in Lean. arXiv:2602.22631, 2026.

Baydin et al. *Automatic Differentiation in Machine Learning: A Survey.* JMLR, 2018.

Boldo & Melquiond. *Flocq: A Unified Library for Proving Floating-Point Algorithms in Coq.* ARITH, 2011.

Bousquet & Elisseeff. *Stability and Generalization.* JMLR, 2002.

Zhang et al. *Efficient Neural Network Robustness Certification with General Activation Functions.* NeurIPS, 2018.