TORCHLEAN: Formalizing Neural Networks in Lean

Robert Joseph George [1] **Jennifer Cruden** [1] **Will Adkisson** [2] **Xiangru Zhong** [3]
Huan Zhang [3] **Anima Anandkumar** [1]

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 treats 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 computation-graph representation. TORCHLEAN supports exact and finite-precision tensor semantics, verified reverse-mode differentiation for supported graph programs, interval and affine bound propagation, CROWN/LiRPA-style certificate checking, import/export workflows, and CUDA-backed execution through explicit FFI boundaries. It also includes formal semantic layers for attention and FlashAttention-style fused operators, state-space sequence models, diffusion and sampling processes, probability kernels, reinforcement-learning objectives and MDPs, and self-supervised objectives. 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 environment. Codebase is released here https://github.com/lean-dojo/TorchLean.

[1]California Institute of Technology [2]Washington University in St. Louis [3]University of Illinois Urbana-Champaign. Correspondence to: Robert Joseph George <rgeorge@caltech.edu>.

The 3rd *AI for Math Workshop* at the 43rd International Conference on Machine Learning (ICML), Seoul, South Korea, 2026. Copyright 2026 by the author(s).

1. Introduction

Neural networks are increasingly embedded in systems where mistakes have physical, scientific, or societal cost. The machine-learning community has therefore placed more weight on guarantees that go beyond empirical accuracy: robustness to perturbations, satisfaction of hard safety or stability constraints, and conservative bounds on quantities derived from a network's computation. This has driven an active verification ecosystem spanning solver-based methods such as Reluplex and Marabou for piecewise-linear networks (Katz et al., 2017; 2019), scalable relaxation and abstract-interpretation methods such as IBP, CROWN, DeepPoly, and related LiRPA approaches (Gowal et al., 2018; Zhang et al., 2018; Singh et al., 2019), and benchmark interfaces such as VNN-COMP (Brix et al., 2024; VNN-COMP, 2024; VNN-LIB, 2021). These methods are powerful, but the guarantee is only as meaningful as the artifact being verified.

In ordinary workflows, the trained network may appear as a PyTorch `nn.Module`, an exported ONNX or FX graph, a compiler IR, a CUDA-backed runtime computation, an external CROWN certificate, or a mathematical object in a proof. These artifacts are connected by exporters, graph rewrites, scalar semantics, layout conventions, preprocessing code, and runtime libraries. If those boundaries are implicit, a property can be proved about one object while the deployed system executes another. TORCHLEAN makes the network definition the semantic reference point: a model is written as a typed Lean program, executed eagerly or in compiled mode, and lowered to an operator-tagged SSA/DAG computation graph whose denotation is shared by execution, automatic differentiation, bound propagation, and certificate checking.

Four recurring failure modes motivate the design. **First,**

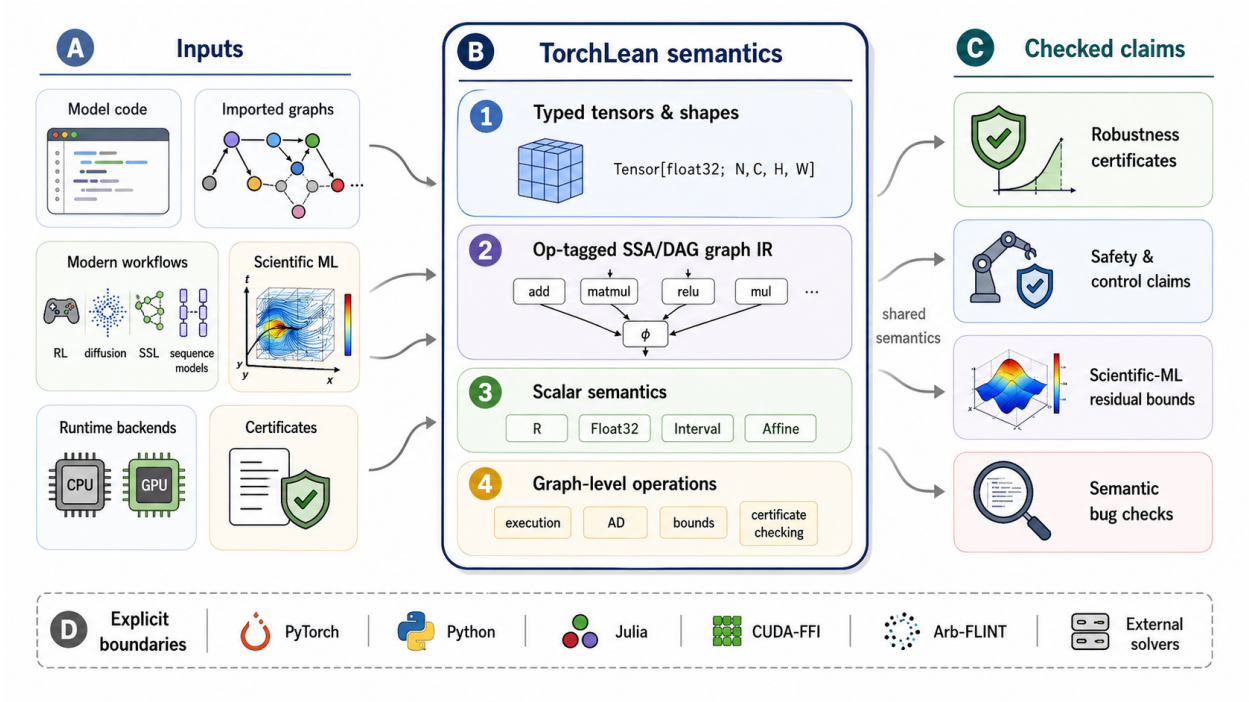

Figure 1. **TORCHLEAN provides a shared semantic layer for verified machine learning.** Model code, imported graphs, modern ML workflows, scientific ML artifacts, runtime backends, and certificates are funneled into typed tensor and operator-tagged SSA/DAG graph semantics in Lean. Execution, AD, bounds, and certificate checking are interpreted against the same representation, while external runtimes and solvers remain explicit boundaries.

export boundaries create semantic drift. Trace-based export can miss control flow and dynamic behavior, and exported graphs inherit opset/version conventions from tools such as ONNX (PyTorch Contributors, 2025; ONNX Contributors, 2026). **Second, floating-point semantics can invalidate idealized guarantees.** Most verifiers state results over reals or simplified numerical models, while deployment uses IEEE-754 arithmetic with rounding, overflow, underflow, NaNs, infinities, signed zeros, mixed precision, and backend-specific algorithms (IEEE Standards Association, 2019; Jia & Rinard, 2020; Szász et al., 2025; Hwang et al., 2025). **Third, certification often stops short of the executed workflow.** Control, autonomy, and scientific-ML pipelines may need guarantees about function values, derivatives, residuals, simulators, or feedback loops, not only static feed-forward logits (Xiang et al., 2018; Raissi et al., 2019). **Fourth, real ML bugs often live at semantic boundaries**: shapes, broadcasting, dtype conversions, padding, masks, train/eval state, preprocessing, and compiler/runtime transformations (Islam et al., 2019; Humbatova et al., 2020; Shen et al., 2021; Wu et al., 2021).

Our contributions are: (i) a PyTorch-style Lean front end for shape-indexed tensors, layers, models, objectives, optimizers, data loaders, and training loops; (ii) a shared operator-tagged SSA/DAG IR with a denotational semantics reused by execution, automatic differentiation, bound propagation, and certificate checking; (iii) explicit scalar semantics spanning exact real tensors, interval and affine abstract domains, proof-oriented FP32/NF rounding models, executable IEEE32 semantics, and trusted runtime/CUDA interfaces; (iv) verified graph-level reverse-mode AD and compiler/IR correspondence for supported fragments; and (v) verification workflows for robustness, PINN residual and derivative bounds, neural-controller safety, VNN-style imported properties, and semantic bug checks over attention, fused kernels, state-space models, diffusion, reinforcement learning, probability kernels, and self-supervised objectives.

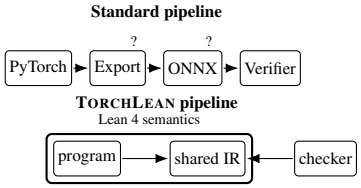

Figure 2. The usual pipeline verifies an exported surrogate. TORCHLEAN uses a single graph semantics for execution and checking.

2. Methodology

Computation graphs and the IR. In TORCHLEAN, a computation graph is a side-effect-free DAG whose nodes are primitive tensor operations and whose edges carry

intermediate tensors. The graph is stored in static-single-assignment form: every intermediate value is defined exactly once and referenced by later nodes. Each node carries an operator tag (e.g. `linear`, `matmul`, `relu`, `conv2d`, `attention`, `softmax`), shape metadata, and scalar-domain metadata. For scalar semantics α and shape s, write $\mathrm{Tensor}_\alpha(s)$ for tensors with entries in α. An operator tag τ has signature

$$\mathrm{sig}(\tau) = (s_1, \ldots, s_k) \to s,$$
$$[\![\tau]\!]_\alpha : \prod_i \mathrm{Tensor}_\alpha(s_i) \to \mathrm{Tensor}_\alpha(s).$$

A well-typed SSA/DAG graph G denotes a total function $[\![G]\!]_\alpha$ obtained by topological evaluation. The same syntax is interpreted over exact reals, executable floating-point domains, intervals, and affine relaxations:

$$[\![G]\!]_{\mathbb{R}}, \quad [\![G]\!]_{\texttt{IEEE32Exec}}, \quad [\![G]\!]_{\texttt{Interval}}, \quad [\![G]\!]_{\texttt{Affine}}.$$

This graph is the common target for training/inference, verified reverse-mode AD, Lean-side bound propagation, and certificate checking. External verifiers are optional producers, not trusted authorities: their outputs are interpreted as finite certificates checked against the Lean denotation of the shared graph.

Specification layer. A tensor is indexed by a scalar domain and a shape. Shapes form an inductive tree, and tensors are represented structurally as total index functions:

$$\mathrm{Tensor}_\alpha(\texttt{scalar}) = \alpha,$$
$$\mathrm{Tensor}_\alpha(\texttt{dim}(n, s)) = \mathrm{Fin}(n) \to \mathrm{Tensor}_\alpha(s).$$

This functional view supports proofs by extensionality: tensor equality reduces to pointwise equality, and maps, zips, reductions, reshapes, broadcasts, masks, and contractions are typed operations rather than silent runtime conventions. At the layer level, a module is a typed morphism $L : \mathrm{Tensor}_\alpha(s_{\mathrm{in}}) \to \mathrm{Tensor}_\alpha(s_{\mathrm{out}})$, with trainable parameters carried by shape-indexed heterogeneous lists. A linear layer denotes

$$\mathrm{Linear}_{W,b}(x)_i = b_i + \sum_{j<d_{\mathrm{in}}} W_{ij} x_j,$$

independent of storage layout. The same definition can be instantiated over $\mathbb{R}$ for proofs, `IEEE32Exec` for explicit finite-precision execution, runtime `Float32`/CUDA for fast paths, or interval/affine domains for verification. Modern operator contracts are represented in the same layer: masked attention is specified as

$$\mathrm{Attn}(Q, K, V, M) = \mathrm{softmax}\left(\frac{QK^\top}{\sqrt{d}} + M\right) V,$$

Table 1. Trust and scalar semantics in TORCHLEAN. The same graph is instantiated in different domains; high-performance native paths are explicit runtime assumptions unless checked or refined.

Mode	Role and trust boundary
$\mathbb{R}$	Exact reference semantics for proofs and graph-level AD.
Interval/affine	Sound enclosure domains for IBP and CROWN/LiRPA.
FP32/NF	Round-on-real proof model with compositional error envelopes.
`IEEE32Exec`	Lean-defined executable binary32 model for core finite paths.
Arb/FLINT	External rigorous ball/interval producer, used through oracle data.
Lean `Float32` / CUDA	Fast execution behind conformance contracts and FFI boundaries.

with the mask part of the formal input semantics. FlashAttention-style fused or tiled implementations are related to this denotation, while sequence/state-space layers, diffusion/sampling maps, RL returns and Bellman operators, and self-supervised view/mask losses are typed contracts over the same tensor language.

Runtime layer. The runtime connects proof-level definitions to runnable training and inference. Users define modules, run forward passes, call backward, and update parameters. Eager mode executes operations immediately and records a dynamic tape containing operator tags, parent references, forward values needed by VJP rules, and cotangent accumulators. Compiled mode lowers the same program to the shared operator-tagged SSA/DAG IR and evaluates that graph. The purpose is semantic rather than only optimization: it produces the graph object consumed by theorems and checkers. For a program p with parameters θ, lowering produces a graph G and parameter store P_θ such that, for the supported fragment and scalar semantics α,

$$p \Downarrow (G, P_\theta) \quad \Longrightarrow \quad \mathrm{eval}_{\mathrm{src}}(p, \theta, x) = [\![G]\!]_\alpha(P_\theta, x).$$

The runtime also includes CSV/NPY and tensor loaders, JSON logs, checkpointing, parameter import/export, and optimizers including SGD, momentum, Adagrad, RMSProp, Adam, AdamW, and Adadelta. Optional CUDA-backed dense/matmul, batched-matmul, reductions, broadcasts, views, convolutions, pooling, FFT/spectral kernels, normalization, attention/fused-attention, gather/scatter, RoPE, and scan-style kernels are exposed through Lean FFI as explicit runtime interfaces, not silently verified code.

Numerical semantics. Many verification claims depend on numerical details that are invisible in a purely real-valued proof. IEEE-754 execution involves rounding, overflow and underflow, NaNs and infinities, signed zeros, subnormals, and backend-specific choices for reductions or fused kernels. In Lean, runtime floating-point types are useful for execution, but their operations are opaque to the kernel. TORCHLEAN therefore separates the fast path from the theorem-facing semantics. Exact real arithmetic is

```python
import torch.nn as nn

# Model + optimizer (semantics live in runtime)
model = nn.Linear(2, 1)
opt = torch.optim.SGD(model.parameters(), lr=0.2)

# Training data
x = torch.tensor([[1.,0.],[0.,1.],[1.,1.]])
y = torch.tensor([[2.],[-3.],[-1.]])

# Training loop (autodiff + updates external)
for step in range(10):
  opt.zero_grad()
  loss = nn.functional.mse_loss(model(x), y)
  loss.backward(); opt.step()
```

```
open TorchLean

-- Model + samples (compiled mode)
def model := sequential [linear 2 1, relu]
def samples := [(x1, y1), (x2, y2), (x3, y3)]

-- Explicit Float32 semantics for this run
def main : IO Unit := do
  let mod <- instantiate model (scalar := IEEE32Exec) .compiled
  let opt := Optim.sgd (scalar := IEEE32Exec) (lr := 0.2)
  for step in [0:10] do
    let loss := mse (mod.forward batch.x) batch.y
    backward loss
    opt.step mod.params
```

Figure 3. **PyTorch vs. TORCHLEAN.** Comparable model/data/loop/update blocks are written in a familiar style, but the TORCHLEAN model can also be lowered to the shared IR, differentiated, and passed to verification.

used for analytic reference statements and graph-level AD. Interval and affine domains are used for verified enclosures. FP32/NF gives a proof-level "compute over reals, then round" semantics for compositional rounding envelopes. `IEEE32Exec` gives an executable binary32 model inside Lean for core arithmetic, including the finite IEEE corner cases needed by checkers. Arb/FLINT is used only as a validated-numerics producer: it may return rigorous ball or interval enclosures for transcendental functions, but the trust boundary is explicit and the returned enclosure is treated as certificate data. CUDA kernels and runtime `Float32` are similarly useful execution paths, but a theorem about them requires a stated conformance contract for layout, rounding mode, denormal policy, reduction order, fusion, and mask conventions.

Modern workflow contracts. The same semantic discipline is applied beyond feed-forward classifiers. Attention layers carry mask contracts and a denotation for scaled dot-product attention; FlashAttention-style fused or tiled kernels are implementations that must agree with that denotation under stated layout and numerical assumptions. State-space and recurrent models are represented by finite recurrences or fixed unrollings when they enter the verifier-facing SSA/DAG fragment. Diffusion and sampling code is described by noising and denoising transitions, schedule parameters, and step-wise sampler maps. RL examples expose rollout tensors, returns, advantages, policies, value functions, finite-MDP views, and simulator boundaries. Self-supervised objectives expose mask/view/target generation and anti-collapse losses. These examples are included in the main system because the failures they catch are semantic failures: a wrong causal mask, an unintended broadcast, an unstable loss, or a mismatched import can invalidate the property even when the program still runs.

Verified reverse-mode AD. Reverse-mode differentiation is proved once at the graph level. Let G be a well-typed SSA/DAG graph over $\mathbb{R}$, let $[\![G]\!]$ be its denotation, and let $\bar{y}$ be an output cotangent. Backpropagation computes the adjoint derivative:

$$\mathrm{backprop}(G, x, \bar{y}) = \big(D[\![G]\!](x)\big)^{*}\bar{y}.$$

Operationally, a reverse topological sweep adds VJP contributions to each parent cotangent. Accumulation is essential: if a value is consumed by multiple downstream nodes, its cotangent is the sum of all downstream contributions. The graph theorem follows by composing local derivative-correctness lemmas for primitive operators; nonsmooth primitives use pointwise hypotheses that expose the corresponding side conditions.

Theorem 2.1: Reverse-mode AD correctness

Let G be a well-typed SSA/DAG computation graph over $\mathbb{R}$ with input context Γ. Assume local derivative correctness for every primitive node. Define $\mathrm{bp}_G(x, seed) := \mathrm{backpropVec}(G, x, seed)$. Then for any input x and output cotangent $seed$,

$$\mathrm{bp}_G(x, seed) = \big(D[\![G]\!](x)\big)^{*} seed.$$

Verification layer. Verification goals are Lean theorems about $[\![G]\!]$: robustness margins, output bounds, invariance/safety constraints, Lyapunov or barrier inequalities, PINN residual bounds, ODE corridor obligations, spline certificate checks, and dataset-backed certified accuracy. Bounds live in an abstract domain $\mathcal{D}$ with concretization γ. A local transformer $\widehat{\tau}$ is sound when $x_i \in \gamma(a_i)$ for all inputs implies $[\![\tau]\!](x_1, \ldots, x_k) \in \gamma(\widehat{\tau}(a_1, \ldots, a_k))$; whole-graph soundness follows by induction over topological order. TORCHLEAN implements a sound IBP core and CROWN/LiRPA-style affine propagation over the same operator denotations.

When stronger tightness is needed, external solvers act as certificate producers. A certificate may supply IBP pre-activation boxes, affine lower/upper bounds, α parameters

for unstable-ReLU relaxations, and optional β phase vectors. Lean checks shapes, operator tags, topological order, local inequalities, phase constraints, parent bounds, and the final property obligation before accepting a theorem of the form

$$\mathrm{checkCert}(G, \Phi, C) = \texttt{true} \quad \Longrightarrow \quad \Phi([\![G]\!]).$$

The verification layer also includes VNN-COMP-style ONNX/VNN-LIB import, graph-CROWN entry points, derivative-residual helpers for PINNs, Lyapunov/CROWN oracle-backed workflows, and regression tests for graph-rewrite or backend drift.

3. Results

We evaluate TORCHLEAN as a semantic ML-systems stack rather than as a replacement for optimized trainers or verifier engines. The central question is whether realistic ML workflows can be connected to typed tensor semantics, graph lowering, finite-precision modes, and checker interfaces without changing the object being analyzed.

Model families and workflow coverage. Figure 4 summarizes the current executable surface. The examples exercise ordinary ML structure: minibatch loaders, multiple epochs, optimizer state, saved parameters, loss curves, sampling loops, rollout tensors, imported weights, and generated certificates. They also exercise the newer parts of the codebase: CUDA/native FFI paths, reductions and broadcasts, views, convolution/pooling and transposed-convolution kernels, FFT/spectral operators, softmax/log-softmax, normalization, attention/fused attention, gather/scatter, positional/RoPE helpers, selective-scan-style sequence kernels, PyTorch/ONNX/VNN-LIB interop, Arb/FLINT validated-numerics calls, Gymnasium/PPO-style RL bridges, and semantic regression tests. These are not leaderboard training claims; they show that verification-oriented semantics can be attached to workflows that look like recognizable ML programs. CNN/FNO examples stress tensor shapes and spectral operators; GPT-style models stress tokenization, causal masking, generation, and attention semantics; Mamba/state-space examples stress recurrence and non-anticipation; diffusion and SSL examples stress masks, views, schedules, latent variables, and reconstruction/prediction losses; RL examples stress rollouts, returns, advantages, policy/value artifacts, and simulator boundaries.

Verification and certificates. The verification stack is evaluated as a checker-oriented system. Native IBP and CROWN/LiRPA-style passes operate over the shared IR. External solvers, Python/Julia scripts, Arb/FLINT, CUDA kernels, and imported ONNX or VNN-LIB files are producers or runtime boundaries. Lean checks the shape, graph, interval, affine, and schema obligations sufficient for the theorem being claimed, or records a named assumption when a boundary is outside the kernel. Using a VNN-COMP-style MNIST-FC slice, Lean-side IBP proves 0/30 instances safe in 6.05s, objective-dependent CROWN proves 6/30 in 22.60s, and CROWN with supplied α parameters proves 6/30 in 14.21s (Table 2). External α-CROWN CPU and CUDA producers prove 13/30 in 8.39s and 7.67s, respectively, but those become TORCHLEAN claims only when their artifacts are checked against the same IR semantics.

Table 2. VNN-style MNIST-FC slice. Checked rows are Lean-verified; CUDA is producer-only unless replayed.

Path	Method	Safe/n	Time
Lean	IBP	0/30	6.05s
Lean	CROWN-Obj	6/30	22.60s
Lean	CROWN+α	6/30	14.21s
CPU prod.	α-CROWN	13/30	8.39s
CUDA prod.	α-CROWN	13/30	7.67s

Certified robustness, control, and PINNs. For a digits linear classifier (`sklearn` digits, $8 \times 8 \to 64$ features, $64 \to 10$, $\varepsilon = 0.02$), the model is nominally correct on 349/360 test inputs and certified robust on 318/360 by replaying an exported bound artifact in Lean and checking the standard logit-margin inequality. The checker averages 0.032ms per example. For neural control, external search proposes a feedback controller $u(x)$ and Lyapunov candidate $V(x)$, while Lean checks CROWN/LiRPA enclosures for V and $\dot{V} = \nabla V \cdot f(x, u(x))$ over regions. Holding Stage 1 weights fixed and using a common Stage 2 baseline, Python Stage 2 takes about 0.015s and yields 9/10 positive-loss candidates; TORCHLEAN Stage 2 under explicit `IEEE32Exec` Float32 semantics takes about 1212s and yields 7/10 positives, while also producing a native CROWN enclosure for the scalar loss. For PINNs, weights trained in Python are loaded into Lean; IBP/CROWN and first/second-derivative bound passes certify residual bounds for smooth operators such as linear layers and `tanh`. These case studies demonstrate a single pattern: high-performance producers and scripts can be useful, but the claim is about the checked graph.

Additional checker-facing case-study details. The three main case studies use the same producer/checker pattern but exercise different parts of the semantics. In the robustness case, the finite artifact is a collection of logit bounds over an ℓ_∞ input box; Lean checks that each bound is compatible with the graph and then proves the margin inequality $z_y > z_k$ for all competing classes. In the controller case, the certificate target is not a classifier margin but a Lyapunov-style inequality over a region: bounds on $V(x)$ and on $\dot{V}(x) = \nabla V(x) \cdot f(x, u(x))$ are

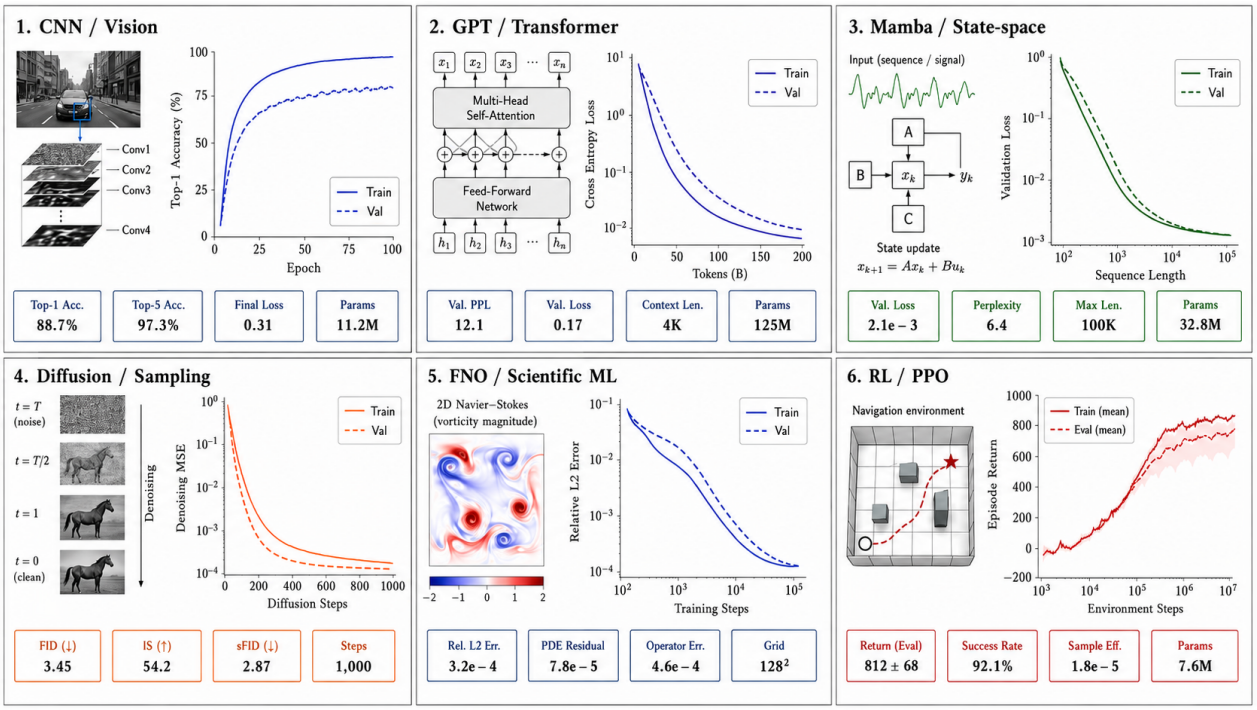

Figure 4. **Representative model families exercised in TORCHLEAN.** Examples cover vision, transformer-style language models, Mamba/state-space sequence models, diffusion/sampling, Fourier-neural-operator and scientific-ML workflows, and reinforcement learning. The formal role is to connect recognizable ML workflows to typed tensor semantics, graph lowering, finite-precision modes, and checker interfaces.

composed with the graph-level derivative semantics and the dynamics contract. In the PINN case, the target is a residual inequality, so the checker combines value bounds with first- and second-derivative interval bounds for the smooth operator subset used in the demo. The point of reporting these together is that the checker does not need a different semantic universe for each task. The graph denotation, node identifiers, scalar semantics, local transfer rules, and final finite inequality are shared; only the property Φ and the certificate schema change.

Semantic bug and numerical stress tests. TORCHLEAN includes regression tests for semantic boundary failures. A bias $b \in \mathbb{R}^C$ should be applied to logits $X \in \mathbb{R}^{B\times C}$ as $Y_{ij} = X_{ij} + b_j$, but a wrong broadcast axis can silently change the function while producing a plausible tensor. A wrong causal mask or stale KV-cache entry can let future-token information influence a current output. A naive log-sum-exp or log-softmax can overflow under Float32, while a stable formulation remains finite. Endpoint evaluation alone is not a sound enclosure method for transcendentals, so Arb-backed enclosure certificates are treated as external oracle data. Directed rounding and signed-zero guards also matter: a ties-to-even addition can be enclosed by `IEEE32Exec` directed endpoints but not by naive runtime `Float32`, and interval division must widen when a denominator contains real zero or signed-zero encodings.

The same tests also exercise the import and runtime boundaries. ONNX/VNN-LIB bundles are not treated as trusted mathematical objects by themselves; they are parsed into typed graph data, and the property matrices, input boxes, and output constraints are checked against the Lean-side representation that is actually used. CUDA, Python, Julia, Gymnasium, and Arb/FLINT enter in the same way: they may produce weights, rollouts, intervals, optimized slopes, or split certificates, but a verification claim is accepted only after the relevant finite obligations are checked for the shared graph. This is the main experimental message of the results section. The examples are deliberately heterogeneous, but the interface they stress is the same: a tensor program becomes an operator-tagged graph, local operators carry their denotations and abstract transformers, scalar semantics are explicit, and the final claim is discharged by a small checker or theorem over that graph.

Across the robustness, PINN, and controller examples, the checked object is always a finite obligation over this graph: a logit-margin inequality, an interval residual bound, a Lyapunov or barrier inequality, or a certificate replay condition. This keeps the experiments comparable even when the surrounding workflow changes from classification to scientific modeling or control. It also makes failures diagnostic: a rejected certificate points to a local operator,

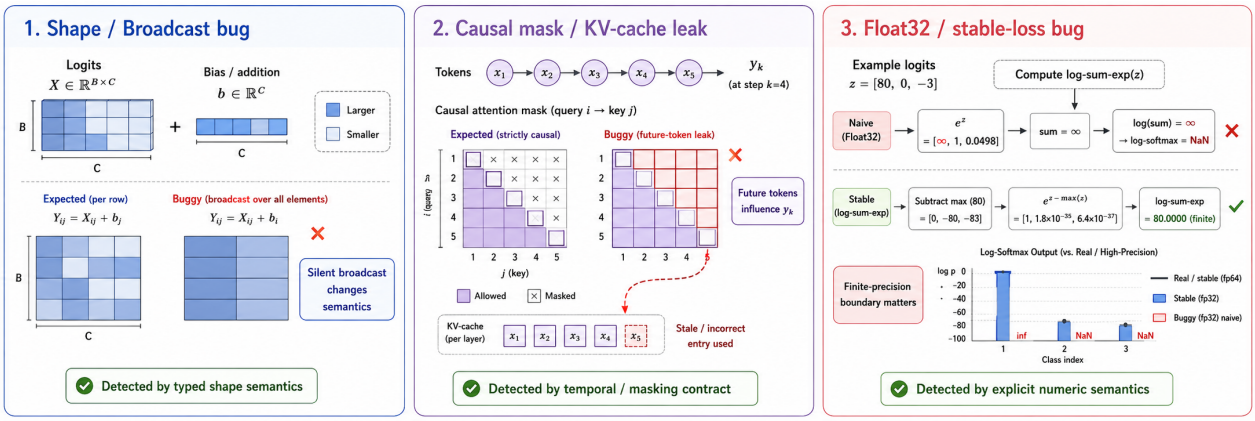

Figure 5. **Representative semantic boundary bugs.** Left: typed shape semantics distinguishes intended row-wise bias addition from unintended broadcasting. Middle: temporal and masking contracts expose causal-mask or KV-cache leaks. Right: explicit finite-precision semantics separates stable log-sum-exp/log-softmax from a naive Float32 implementation that can overflow or produce NaNs.

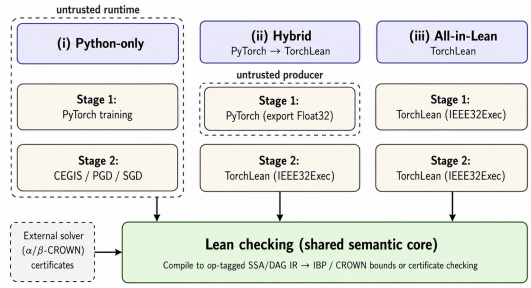

Figure 6. Neural-controller workflow in three execution settings. Certification claims are checked against the shared IR semantics; external optimizers and runtime systems are producers or execution paths.

shape, numeric, or schema obligation rather than to an opaque verifier/runtime mismatch.

4. Related Work

Neural network verification (solver-based and bound-propagation). Neural network verification is central in safety and robustness analysis (Xiang et al., 2018; Katz et al., 2019; Zhang et al., 2018; Xu et al., 2021). Broadly, the literature spans (i) *solver-based* methods such as Reluplex and Marabou (Katz et al., 2019), which encode verification problems as satisfiability/constraint queries, and (ii) *abstract-interpretation / relaxation* methods such as IBP (Gowal et al., 2018) and linear bound propagation (CROWN/DeepPoly and related abstractions) (Dvijotham et al., 2018; Gehr et al., 2018; Raghunathan et al., 2018; Singh et al., 2018; 2019; Wang et al., 2018; Wong & Kolter, 2018; Zhang et al., 2018). These tools often operate on exported artifacts (ONNX/TorchScript/custom IRs) and therefore inherit an additional trust boundary at the export/interpretation step. This pipeline is standardized and stress-tested in VNN-COMP, where instances are packaged as an ONNX network together with a VNN-LIB property specification (Brix et al., 2024; VNN-COMP, 2024; VNN-LIB, 2021), making the export/interface boundary an explicit part of the evaluation regime.

CROWN-family optimizers and certificate checking. CROWN-family methods derive output bounds by propagating sound linear relaxations of nonlinearities and viewing the result through a dual objective; α-CROWN (Xu et al., 2021) tightens bounds by optimizing relaxation parameters, and β-CROWN integrates splitting/branch-and-bound to further tighten bounds (Wang et al., 2021). TORCHLEAN implements a Lean-based CROWN/LiRPA core over our shared operator-tagged IR semantics (proved-sound IBP, a basic forward affine pass, and an objective-dependent backward/dual pass). Reimplementing the full α/β-CROWN optimization stack inside Lean including the parameter-optimization heuristics and branch-and-bound search is ongoing work. When strongest current tightness is needed, we instead treat external solvers as untrusted producers and check their exported bounds/certificates against the same IR semantics, keeping the trusted computing base to a small checker.

Formalization in theorem provers and Lean infrastructure. ITP's have been used to formalize ML-relevant mathematics (e.g., generalization bounds (Bagnall & Stewart, 2019), activation-function libraries (Aleksandrov & Völlinger, 2023), and network translations (Gummersbach et al., 2025)). Lean's ecosystem (notably `mathlib`) makes these developments practical, but full pipeline neural verification remains hard because it requires tensor representations that scale to modern architectures, a sound differentiation story, and numerical semantics that are explicit enough to connect proofs to execution. `TensorLib` provides a verified tensor library for Lean (leanprover, 2025) and, as of this writing, is under active development.

SciLean, verified floats, and complementary developments. SciLean is a Lean-based library for scientific computing with array abstractions and automatic differentiation (Contributors, 2023). TORCHLEAN targets a different point in the design space: typed tensor computation graphs with shape-indexed tensors, a graph-parametric reverse-mode correctness theorem for SSA/DAGs, and certificate-checked verification workflows. On numerics, verified floating-point frameworks such as Flocq (Boldo & Melquiond, 2011) motivate the separation we use between rounding models for proofs (FP32/NF) and an executable IEEE-style kernel (`IEEE32Exec`), with an explicit trust boundary to runtime floats. HopfieldNet formalizes Hopfield/Boltzmann-style energy arguments in Lean (Cipollina et al., 2025). The Hopfield case study in TORCHLEAN uses the same tensor and graph semantics to state the standard energy-decrease and convergence-style results.

5. Discussion

TORCHLEAN is best understood as a semantics layer for neural network systems rather than as a replacement for existing training frameworks or specialized verifiers. The main design choice is to make the network definition the semantic reference point and to connect execution, differentiation, bound propagation, certificate checking, and external artifacts to a shared operator-tagged SSA/DAG IR. This addresses a common source of fragility in verified ML workflows: the artifact that is trained, exported, optimized, or executed is often not exactly the artifact that is analyzed. In TORCHLEAN, this boundary is made explicit. Imported ONNX/VNN-LIB files, PyTorch or Julia producers, Arb/FLINT enclosures, CROWN-family certificates, Gymnasium rollouts, and CUDA/native kernels are treated as producers or runtime interfaces whose artifacts must either be checked in Lean or recorded as named assumptions. This separation is also important for modern ML workflows. Verification is no longer only about small feed-forward classifiers. Attention masks, FlashAttention style fused operators, state-space scans, diffusion schedules, self-supervised views, RL returns, and scientific ML residuals all carry mathematical contracts that can be broken by small implementation choices. TORCHLEAN makes these contracts visible as typed specifications. The current system therefore supports two complementary uses: proving semantic properties of supported graph fragments, and detecting semantic boundary failures such as shape/broadcasting mistakes, causal-mask leaks, unstable losses, export drift, and certificate inconsistencies.

The main limitations are coverage and conformance. The verified fragment is necessarily smaller than the full surface of PyTorch, CUDA, and modern ML libraries. Extending TORCHLEAN requires adding operator semantics, local derivative rules, abstract transformers, finite-precision models, and certificate checkers for new primitives. Native CUDA execution is useful for scale, but it remains outside Lean's kernel; connecting it more tightly to the formal semantics requires stronger conformance testing, verified wrappers, or eventually verified kernels for important primitives. Similarly, external certificate producers are useful for tightness and performance, but the formal guarantee comes only from what the Lean checker validates. These limitations are deliberate: the system favors explicit boundaries over implicit trust.

6. Conclusion

TORCHLEAN advances a semantic approach to verified machine learning. Models are written as executable neural network programs, lowered to a shared operator-tagged SSA/DAG IR, and interpreted by execution, automatic differentiation, bound propagation, and certificate checking through the same formal semantics. This makes it possible to state machine-checked claims about robustness, control-oriented safety, scientific ML residuals, VNN-style imported artifacts, and semantic regression tests without relying on an informal export pipeline. The framework combines a PyTorch-style Lean API, typed tensors and layers, verified graph-level differentiation, explicit floating-point semantics, native IBP/CROWN style verification, certificate checking, optional CUDA-backed execution through Lean FFI, and interop with PyTorch, Python, Julia, Arb/FLINT, Gymnasium, and CROWN-family tools. The contribution is not that every external producer or runtime is verified inside Lean. Rather, TORCHLEAN makes clear which claims are checked by Lean, which artifacts are replayed or validated, and which runtime or oracle assumptions remain outside the kernel.

The broader value of this approach is that it gives multiple communities a shared semantic object to build on. Verification researchers can check certificates against executable models; ML systems researchers can study export, compilation, numerical, and backend boundaries with formal contracts; and scientific ML, control, and robotics researchers can connect learned components to residual, safety, or stability claims without losing the semantics of the implemented system. Verified ML also needs shared infrastructure in the spirit of what PyTorch provided for empirical ML: a substrate where models, tools, transformations, and artifacts can be composed. Lean has demonstrated this model through `mathlib`, and emerging libraries such as CSLib point toward similar foundations for verified software. TORCHLEAN contributes to this ecosystem by connecting neural-network computation, formal semantics, and certificate based verification in one theorem-proving environment.

Acknowledgments

Robert Joseph George is supported by DARPA expMath and AA2.AIMING-1-NSF.2522494 (AIMing: Synergistic Advancement of AI and Mathematics with Dr. Sergei Gukov). Jennifer Cruden and Will Adkisson are supported by Caltech SURF fellowships. Huan Zhang and Xiangru Zhong are supported in part by NSF IIS-2331967 and the AI2050 program at Schmidt Sciences. Anima Anandkumar is supported in part by the Bren endowed chair, ONR MURI grant N00014-18-1-2624, DARPA expMath, and the AI2050 Senior Fellowship program at Schmidt Sciences.

References

Aleksandrov, A. and Völlinger, K. Formalizing piecewise affine activation functions of neural networks in coq, 2023. URL https://arxiv.org/abs/2301.12893.

Assran, M., Duval, Q., Misra, I., Bojanowski, P., Vincent, P., Rabbat, M., LeCun, Y., and Ballas, N. Self-supervised learning from images with a joint-embedding predictive architecture. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 15619–15629, 2023.

Bagnall, A. and Stewart, G. Certifying the true error: Machine learning in coq with verified generalization guarantees. *Proceedings of the AAAI Conference on Artificial Intelligence*, 33(01):2662–2669, 2019. doi: 10.1609/aaai.v33i01.33012662. URL https://ojs.aaai.org/index.php/AAAI/article/view/4115.

Bardes, A., Ponce, J., and LeCun, Y. Vicreg: Variance-invariance-covariance regularization for self-supervised learning. In *International Conference on Learning Representations*, 2022.

Bellman, R. *Dynamic Programming*. Princeton University Press, 1957.

Boldo, S. and Melquiond, G. Flocq: A unified library for proving floating-point algorithms in Coq. In *Proceedings of the 20th IEEE Symposium on Computer Arithmetic (ARITH)*, 2011. doi: 10.1109/ARITH.2011.40. URL https://dl.acm.org/doi/10.1109/ARITH.2011.40.

Brisebarre, N., Hanrot, G., Muller, J.-M., and Zimmermann, P. Correctly rounded evaluation of a function: Why, how, and what cost? *ACM Computing Surveys*, 2025. doi: 10.1145/3747840.

Brix, C., Bak, S., Johnson, T. T., and Wu, H. The fifth international verification of neural networks competition (vnn-comp 2024): Summary and results, 2024.

Cipollina, M., Karatarakis, M., and Wiedijk, F. Formalized hopfield networks and boltzmann machines, 2025. URL https://arxiv.org/abs/2512.07766.

Contributors, S. Scilean: Scientific computing in lean. Software, 2023. URL https://github.com/lecopivo/SciLean.

Cybenko, G. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems*, 2:303–314, 1989. doi: 10.1007/BF02551274.

Dao, T., Fu, D. Y., Ermon, S., Rudra, A., and Ré, C. Flashattention: Fast and memory-efficient exact attention with io-awareness. In *Advances in Neural Information Processing Systems*, volume 35, pp. 16344–16359, 2022.

Dvijotham, K., Stanforth, R., Gowal, S., Mann, T., and Kohli, P. A dual approach to scalable verification of deep networks. In *Proceedings of the 34th Conference on Uncertainty in Artificial Intelligence (UAI 2018)*, 2018. URL https://auai.org/uai2018/proceedings/papers/204.pdf. arXiv:1803.06567.

Flocq Developers. Flocq: Theoretical background (floats for coq). https://flocq.gitlabpages.inria.fr/theos.html, 2025.

Gehr, T., Mirman, M., Drachsler-Cohen, D., Tsankov, P., Chaudhuri, S., and Vechev, M. Ai2: Safety and robustness certification of neural networks with abstract interpretation. In *2018 IEEE symposium on security and privacy (SP)*, pp. 3–18. IEEE, 2018.

Gowal, S., Dvijotham, K., Stanforth, R., Bunel, R., Qin, C., Uesato, J., Arandjelović, R., Mann, T., and Kohli, P. On the effectiveness of interval bound propagation for training verifiably robust models. In *NeurIPS Workshop on Security in Machine Learning*, 2018. URL https://arxiv.org/abs/1810.12715.

Gummersbach, L. A., Völlinger, K., and Aleksandrov, A. A formally verified neural network converter for the interactive theorem prover coq. In Rümmer, P. and Wu, Z. (eds.), *Theoretical Aspects of Software Engineering*, pp. 197–214, Cham, 2025. Springer Nature Switzerland.

He, K., Chen, X., Xie, S., Li, Y., Dollár, P., and Girshick, R. Masked autoencoders are scalable vision learners. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 16000–16009, 2022.

Ho, J., Jain, A., and Abbeel, P. Denoising diffusion probabilistic models. In *Advances in Neural Information Processing Systems*, volume 33, pp. 6840–6851, 2020.

Hornik, K. Approximation capabilities of multilayer feedforward networks. *Neural Networks*, 4(2):251–257, 1991. doi: 10.1016/0893-6080(91)90009-T.

Humbatova, N., Jahangirova, G., Bavota, G., Riccio, V., Stocco, A., and Tonella, P. Taxonomy of real faults in deep learning systems. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE 2020, pp. 1110–1121, New York, NY, USA, 2020. Association for Computing Machinery. doi: 10.1145/3377811.3380395.

Hwang, G., Lee, W., Park, Y., Park, S., and Saad, F. Floating-point neural networks are provably robust universal approximators. In *Computer Aided Verification (CAV 2025)*, volume 15932 of *Lecture Notes in Computer Science*, pp. 301–326. Springer, 2025. doi: 10.1007/978-3-031-98679-6_14. URL https://doi.org/10.1007/978-3-031-98679-6_14. Full version: arXiv:2506.16065.

IEEE Standards Association. IEEE Standard for Floating-Point Arithmetic (IEEE Std 754-2019). https://standards.ieee.org/standard/754-2019.html, 2019.

Islam, M. J., Nguyen, G., Pan, R., and Rajan, H. A comprehensive study on deep learning bug characteristics. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, pp. 510–520, New York, NY, USA, 2019. Association for Computing Machinery. doi: 10.1145/3338906.3338955.

Jia, K. and Rinard, M. C. Exploiting verified neural networks via floating point numerical error. *arXiv preprint arXiv:2003.03021*, 2020. URL https://arxiv.org/abs/2003.03021.

Katz, G., Barrett, C., Dill, D. L., Julian, K., and Kochenderfer, M. J. Reluplex: An efficient SMT solver for verifying deep neural networks. In Majumdar, R. and Kuncak, V. (eds.), *Computer Aided Verification (CAV 2017)*, volume 10426 of *Lecture Notes in Computer Science*, pp. 97–117. Springer, July 2017. doi: 10.1007/978-3-319-63387-9_5. URL http://theory.stanford.edu/~barrett/pubs/KBD+17.pdf.

Katz, G., Huang, D. A., Ibeling, D., Julian, K., Lazarus, C., Lim, R., Shah, P., Thakoor, S., Wu, H., Zeljić, A., Dill, D. L., Kochenderfer, M. J., and Barrett, C. The marabou framework for verification and analysis of deep neural networks. In *Computer Aided Verification (CAV 2019)*, 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.

Lean Prover Community. Floating-point numbers (lean reference manual), 2025. URL https://lean-lang.org/doc/reference/latest/Basic-Types/Floating-Point-Numbers/.

leanprover. Tensorlib: A verified tensor library in lean. GitHub repository, 2025. URL https://github.com/leanprover/TensorLib.

ONNX Contributors. ONNX versioning documentation (ir and operator set versioning). https://onnx.ai/onnx/repo-docs/Versioning.html, 2026.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al. PyTorch: An imperative style, high-performance deep learning library. *Advances in Neural Information Processing Systems*, 32, 2019. URL https://proceedings.neurips.cc/paper_files/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html.

Puterman, M. L. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley, 1994.

PyTorch Contributors. torch.onnx — pytorch documentation (limitations of trace-based onnx export and numeric differences across runtimes). https://docs.pytorch.org/docs/stable/onnx, 2025.

Raghunathan, A., Steinhardt, J., and Liang, P. Certified defenses against adversarial examples. In *International Conference on Learning Representations (ICLR)*, 2018. URL https://openreview.net/forum?id=Bys4ob-Rb. arXiv:1801.09344.

Raissi, M., Perdikaris, P., and Karniadakis, G. E. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational Physics*, 378:686–707, 2019. doi: 10.1016/j.jcp.2018.10.045.

Schulman, J., Moritz, P., Levine, S., Jordan, M., and Abbeel, P. High-dimensional continuous control using generalized advantage estimation. In *International Conference on Learning Representations*, 2016.

Shen, Q., Ma, H., Chen, J., Tian, Y., Cheung, S.-C., and Chen, X. A comprehensive study of deep learning compiler bugs. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2021, pp. 968–980,

New York, NY, USA, 2021. Association for Computing Machinery. doi: 10.1145/3468264.3468591.

Singh, G., Gehr, T., Mirman, M., Püschel, M., and Vechev, M. Fast and effective robustness certification. In *Advances in Neural Information Processing Systems*, volume 31, pp. 10825–10836, 2018. URL https://papers.neurips.cc/paper/2018/hash/f2f446980d8e971ef3da97af089481c3-Abstract.html.

Singh, G., Gehr, T., Püschel, M., and Vechev, M. An abstract domain for certifying neural networks. *Proceedings of the ACM on Programming Languages*, 3(POPL):41:1–41:30, 2019. doi: 10.1145/3290354. URL https://dl.acm.org/doi/10.1145/3290354.

Song, J., Meng, C., and Ermon, S. Denoising diffusion implicit models. In *International Conference on Learning Representations*, 2021a.

Song, Y., Sohl-Dickstein, J., Kingma, D. P., Kumar, A., Ermon, S., and Poole, B. Score-based generative modeling through stochastic differential equations. In *International Conference on Learning Representations*, 2021b.

Sutton, R. S. and Barto, A. G. *Reinforcement Learning: An Introduction*. MIT Press, 2 edition, 2018.

Szász, A., Bánhelyi, B., and Jelasity, M. No soundness in the real world: On the challenges of the verification of deployed neural networks, 2025. URL https://arxiv.org/abs/2506.01054.

VNN-COMP. International verification of neural networks competition, 2024. URL https://vnn-comp.github.io/.

VNN-LIB. A standard for neural network verification specifications, 2021. URL https://www.vnnlib.org/.

Wang, S., Pei, K., Whitehouse, J., Yang, J., and Jana, S. Efficient formal safety analysis of neural networks. *Advances in Neural Information Processing Systems*, 31, 2018.

Wang, S., Zhang, H., Xu, K., Lin, X., Jana, S., Hsieh, C.-J., and Kolter, Z. Beta-crown: Efficient bound propagation with per-neuron split constraints for neural network robustness verification. In *Advances in Neural Information Processing Systems*, 2021.

Wong, E. and Kolter, J. Z. Provable defenses against adversarial examples via the convex outer adversarial polytope. In *International Conference on Machine Learning*, pp. 5286–5295. PMLR, 2018.

Wu, D., Shen, B., and Chen, Y. An empirical study on tensor shape faults in deep learning systems, 2021. URL https://arxiv.org/abs/2106.02887.

Xiang, W., Musau, P., Wild, A. A., Manzanas Lopez, D., Hamilton, N., Yang, X., Rosenfeld, J., and Johnson, T. T. Verification for machine learning, autonomy, and neural networks survey. *arXiv preprint arXiv:1810.01989*, 2018. doi: 10.48550/arXiv.1810.01989.

Xu, K., Zhang, H., Wang, S., Wang, Y., Jana, S., Lin, X., and Hsieh, C.-J. Fast and complete: Enabling complete neural network verification with rapid and massively parallel incomplete verifiers. In *International Conference on Learning Representations*, 2021.

Yarotsky, D. Error bounds for approximations with deep relu networks. *Neural Networks*, 94:103–114, 2017. doi: 10.1016/j.neunet.2017.07.002.

Zbontar, J., Jing, L., Misra, I., LeCun, Y., and Deny, S. Barlow twins: Self-supervised learning via redundancy reduction. In *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pp. 12310–12320, 2021.

Zhang, H., Weng, T.-W., Chen, P.-Y., Hsieh, C.-J., and Daniel, L. Efficient neural network robustness certification with general activation functions. *Advances in Neural Information Processing Systems*, 31:4939–4948, 2018. URL https://arxiv.org/abs/1811.00866.

Zimmermann, P. Why transcendentals and arbitrary precision? https://members.loria.fr/PZimmermann/talks/why.pdf, 2005.

A. Appendix Overview

Appendix guide

- **Appendix B: TORCHLEAN frontend and graph semantics.** Program modes, module API, lowering to the shared IR, graph-level AD, node correctness, and the native runtime/FFI boundary.
- **Appendix C: Modern workflow semantics.** Attention and FlashAttention contracts, RL/MDP semantics, diffusion and probability layers, SSL objectives, and model-family contracts.
- **Appendix D: Numerical semantics.** IEEE-style execution, rounded-real models, interval/enclosure semantics, Arb/FLINT oracles, and hardware/runtime trust assumptions.
- **Appendix E: Verification and certificates.** IBP, CROWN/LiRPA, certificate schemas, VNN-COMP-style import, controller/PINN checks, and worked examples.
- **Appendix F: Approximation theorems.** Universal approximation over reals and the Float32 execution direction.
- **Appendix G: Limitations and discussion.** Coverage limits, runtime conformance gaps, and future extensions.

Reading conventions

Proved means a Lean theorem over the stated semantics. **Checked** means a Lean executable checker validates a finite artifact or recomputes a local obligation. **Boundary** means an external runtime, solver, simulator, or numerical producer is used through an explicit assumption or conformance contract rather than silently trusted.

B. TORCHLEAN: a PyTorch-style front end with a single semantic target

TORCHLEAN is our user-facing interface for building and training neural networks. It provides a PyTorch-like programming model where users write models using familiar operations (linear layers, convolutions, activations), but with a crucial difference: the same model definition can execute in two modes that share a unified semantic foundation. This design eliminates the semantic gap between training code and verification analysis.

B.1. Core design: unified semantics for execution and analysis

Our core design choice is to treat the computation graph as the semantic target for *both* execution and reasoning. In TORCHLEAN, programs lower to a typed, operator-tagged graph IR: nodes are primitive operators (e.g., matrix multiplication or ReLU) and edges carry tensor values. We store graphs in SSA/DAG form: each intermediate value is defined exactly once (SSA), and the dataflow graph is acyclic (DAG). These constraints are standard compiler discipline, but they are also exactly what we want here: evaluation is deterministic by a topological order, and proofs can proceed by induction over that order while reusing the same operator semantics that the runtime executes.

In TORCHLEAN, the SSA/DAG representation serves multiple purposes simultaneously:

- **Execution:** The graph can be evaluated to compute forward values and gradients.
- **Formal reasoning:** The graph has a precise mathematical denotation $[\![G]\!]$ that we can state theorems about.
- **Verification:** Bound propagation algorithms operate directly on the graph structure.

This unified representation is what makes the system work: there is no "export step" that might introduce semantic drift between the model as written and the model as analyzed.

The invariant used by the rest of the section The important invariant is simple: every runtime artifact that enters a theorem has a graph node, a shape, and a denotation. If a node id is i, then the value table produced by execution, the cotangent table produced by reverse mode, and the bound table produced by verification all refer to the same i. This prevents a common failure mode in mixed systems: the proof talks about one tensor while the runtime or verifier silently talks about another. The invariant can be summarized as

$$\forall i < |G|, \qquad v_i = [\![G_i]\!]_\alpha(\rho) \quad \text{and} \quad v_i \in \gamma(B_i)$$

whenever evaluation and bound propagation both succeed. Here ρ is the input and parameter environment, B_i is the abstract value stored for node i, and γ is the concretization map for the chosen abstract domain. The same statement is used in different ways: execution establishes the left equality, IBP or CROWN establishes the right containment, and certificate checking recomputes the finite obligations that connect the two.

Listing 1. Schematic public model definition through the stable facade.

Table 3. Verified scope in TORCHLEAN (summary). "Proved" means a Lean theorem over the stated semantics; "Checked" means a small executable validator that re-computes or validates constraints; "Assumed/External" marks trust boundaries or future work. Appendix E also tabulates operator level verification coverage.

Component	What the paper claims (scope)	Status	Where
Shared IR semantics	Operator tagged SSA/DAG IR with a precise denotation reused by execution and verification.	Lean-defined semantics	Sections 2, B
Reverse-mode AD	Backpropagation equals the adjoint Fréchet derivative for well-typed SSA/DAG graphs over $\mathbb{R}$, assuming local per-op derivative correctness (with a pointwise variant for non-smooth ops).	Proved	Theorem 2.1, Appendix B
TORCHLEAN compilation	"Compiled mode" produces the verifier IR; full compiler correctness for arbitrary TORCHLEAN programs is not yet proved.	Proved (fragment) / Future (full)	Theorem B.3
IBP / CROWN operators	Sound transfer rules and relaxations for a curated set of operators; additional ops use conservative fallbacks or are treated as unsupported in the strongest theorems.	Mixed (proved subset)	Appendix E
Certificate checking	Small Lean checkers validate certificate structure and the final enclosure constraints needed to discharge a theorem about $[\![G]\!]$.	Checked (plus proved core)	Section 3, Appendix E
External solvers (e.g. α/β-CROWN)	Used only as untrusted certificate producers; full optimizer internals are outside the checked certificate scope in this work.	Assumed/External	Section 3, Appendix E
Float32 semantics	Executable IEEE-754 binary32 model (`IEEE32Exec`) and proof-level rounding models (FP32/NF), with an internal refinement on finite executions.	Proved (internal)	Section 1, Appendix D
Hardware Float32 link	Connecting deployed `Float32` hardware/runtime behavior to `IEEE32Exec` is target-specific and not discharged in this work.	Assumed/External	Section 6, Appendix D.1

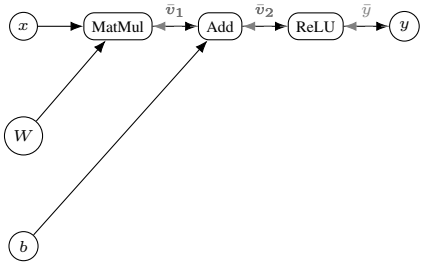

Figure 7. Example SSA/DAG computation graph: $y = \mathrm{ReLU}(Wx + b)$. Each intermediate value (v_1, v_2) is assigned once. Forward pass (solid arrows) evaluates nodes in topological order. Backward pass (dashed arrows) propagates gradients in reverse topological order, accumulating at each node.

```
import NN.API.Public
open NN

-- One model definition is used by
   execution, compilation, and checking.
def tiny : Spec.Model :=
  Linear (in := 4) (out := 8)
    |>.then ReLU
    |>.then (Linear (in := 8) (out := 1))

#eval tiny.run (Tensor.ones [1, 4])
```

Listing 1 shows the intended surface shape: the user writes a small model once, then chooses an execution or verification view. The appendix code listings use the same convention. They are schematic when the exact repository module names are not part of the paper claim; the theorem statements remain about the graph semantics, not about a particular file layout.

Listing 2. Schematic IR objects shared by evaluation and verification.

```
inductive OpKind where
| input | const | linear | matmul | add |
    relu | tanh | softmax
| conv2d | flatten | reduceSum | mseLoss

structure Node where
  id       : Nat
  kind     : OpKind
  parents  : List Nat
  outShape : Shape

structure Graph where
  nodes     : Array Node
  inputIds  : List Nat
  outputIds : List Nat

-- SSA/DAG well orderedness: parents of
    node i have smaller ids.
def wellOrdered (G : Graph) : Prop :=
  forall n, n in G.nodes.toList ->
    forall p, p in n.parents -> p < n.id
```

This IR is intentionally small. Adding a new primitive starts by adding a tag, a shape rule, and a denotation for that tag. Only after that do we add derivative rules, interval rules, affine relaxations, or native runtime implementations.

B.2. Execution modes and compilation to IR

TORCHLEAN supports two execution modes that share the same semantic foundation but differ in when and how the computation graph is constructed. **Eager mode (imperative, tape-based).** In eager mode, operations execute immediately as you call them, similar to PyTorch's default behavior. As operations run, they are recorded into a *computation tape*: a dynamic data structure that stores the sequence of operations and their dependencies. This tape is built incrementally during the forward pass, and then used during the backward pass to compute gradients. The key property of eager mode is that it matches the familiar "define-by-run" workflow: you write code that looks like ordinary function calls, and the tape is constructed implicitly as a side effect. This makes debugging straightforward (you can inspect intermediate values immediately) and allows dynamic control flow (if statements, loops with data-dependent bounds).

Eager mode (tape semantics vs. graph semantics). Operationally, eager execution builds a tape on the fly; semantically, each completed run induces a well-typed SSA/DAG graph G that matches the run's forward values, and whose backpropagation matches the tape's backward pass. This correspondence is the bridge that lets us develop and debug in eager mode while still applying graph-level theorems (autograd correctness, bounds, and certificate checking) to the same semantic target (Theorem B.2).

Compiled mode (static graph). In compiled mode, the TORCHLEAN program is analyzed upfront to construct a static SSA/DAG graph representation. This is "compilation" only in the sense of *lowering* to a formal IR for reasoning and verification; it is not intended to compete with `torch.compile`-style kernel fusion or hardware-level optimization. The compiled graph is exactly the object that formal theorems reason about: it has a precise denotation $[\![G]\!]$, and we can state and prove correctness properties about it. The advantage of compiled mode is that the graph structure is explicit and available for analysis before execution, so bound propagation algorithms, certificate checkers, and proofs all operate on the same graph object that the runtime evaluates. See Appendix B.8 for details.

B.3. Native runtime and Lean FFI boundary

The runtime layer includes optional native execution paths implemented through Lean's FFI. These paths are useful for larger examples, but they are intentionally separated from theorem statements. At the Lean level, a native call is an opaque or external function with checked metadata around it; at the C/CUDA level, it is ordinary systems code that manages memory and calls kernels. The purpose of the boundary is to keep the semantic claim precise: Lean proves facts about the graph/operator denotation, and native execution is admitted only through an explicit agreement contract.

A typical primitive therefore has three pieces:

1. a **specification**, such as a tensor operator $[\![\tau]\!]_\alpha$;
2. a **runtime implementation**, such as a CPU, CUDA, cuBLAS, or cuFFT path;
3. a **conformance condition**, stating when the runtime result agrees with the specified finite-precision semantics.

For example, a CUDA matrix multiplication call is not itself a proof of matrix-multiplication correctness. The formal statement is conditional on a deployment contract such as

$$\mathsf{cudaBmm}(A, B) =_{\mathsf{tol}} [\![\mathtt{bmm}]\!]_{\mathtt{Float32}}(A, B),$$

where $=_{\mathsf{tol}}$ may be exact bit equality for a deterministic path or an explicit tolerance/conformance relation for a tested backend. The contract also records shape, dtype, layout, device, and determinism assumptions.

Listing 3. Schematic checked call pattern for a native backend.

```
/-- The theorem-facing operator is still
    the spec operator. -/
def matmulSpec (A B : Tensor Float32 s) :
    Tensor Float32 t := ...

/-- The runtime path is an optional
    implementation boundary. -/
def matmulCudaChecked (A B : RuntimeTensor)
     : IO RuntimeTensor := do
  checkShape A.shape expectedLeft
  checkShape B.shape expectedRight
  checkDevice A.device .cuda
  Cuda.Buffer.bmm A.buffer B.buffer
```

This is also why CUDA examples in the paper are phrased as accelerated executions or conformance-tested runtime paths, not as verified CUDA kernels. The value of adding the FFI path is practical: it makes longer runs, larger tensors, FNO/spectral examples, and attention-style examples feasible while preserving the formal boundary that the checker relies on.

Compilation process. When you write a TORCHLEAN program, it's a backend-generic definition that can run in different modes. But for verification, we need a concrete graph structure that we can reason about formally. The compilation process transforms a TORCHLEAN program into an operator-tagged IR graph. Here's how it works: the compiler walks through the program structure, and for each operation (like `linear`, `relu`, `softmax`), it creates a corresponding IR node with the appropriate operation kind and shape information. Parameters are stored separately in a parameter store, and the graph structure captures the data

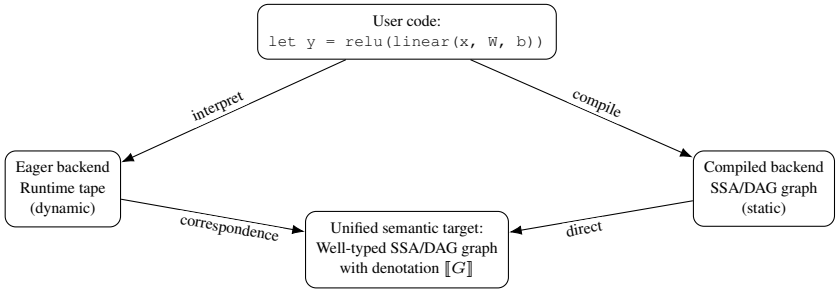

Figure 8. Two execution paths to a unified semantic target. The same TORCHLEAN program can be interpreted eagerly (building a tape dynamically) or compiled (building a static graph upfront). Both paths produce computations that share the same well-typed SSA/DAG denotation, enabling unified reasoning about execution and verification.

dependencies between operations. For the forward-only fragment underlying our verification demos, we make this precise by proving a semantics-preservation theorem: the compiled IR evaluator agrees with the fragment's source evaluator for all inputs and parameters.

The proof is easier to read if the compiler is separated into two maps. The first map extends the graph with new nodes and returns their ids; the second map interprets the generated graph. The preservation theorem then states that the source evaluator and the graph evaluator build the same tensor, or fail for the same explicit reason. This exception-valued form is useful in Lean because it keeps shape errors visible instead of hiding them under partial functions.

Listing 4. Schematic lowering interface used by the forward fragment.

```
inductive EvalError where
| missingParent | shapeMismatch |
    unsupportedOp

structure CompileState where
  graph  : Graph
  params : ParamStore

def emitNode (st : CompileState) (k :
    OpKind)
     (parents : List Nat) (s : Shape) : Prod
      CompileState Nat :=
  -- returns the updated graph and the
    fresh SSA id
  ...

def lowerForward (p : ForwardProg) (st :
    CompileState) :
     Except EvalError (Prod CompileState Nat
    ) :=
  match p with
  | .input j        => pure (st, st.graph.
    inputIds[j]!)
  | .linear W b x   => do
      let (st, xid) <- lowerForward x st
      let (st, wid) <- emitNode st .const
    [] W.shape
      let (st, bid) <- emitNode st .const
    [] b.shape
      let (st, mid) <- emitNode st .matmul
    [wid, xid] (linearOut W)
      emitNode st .add [mid, bid] (
    linearOut W)
  | .relu x         => do
      let (st, xid) <- lowerForward x st
      emitNode st .relu [xid] (shapeOf x)
```

The actual development uses the repository's concrete program and graph datatypes, but the proof obligation has this shape: every emitted node uses only earlier node ids, every emitted shape matches the operator contract, and the evaluator for the emitted node agrees with the source operation at the corresponding program point.

Proved forward compiler correctness. We isolate a first-order, SSA-style forward fragment and prove that lowering it to the operator-tagged IR preserves semantics. The theorem is stated over *exception-valued* evaluation: both the source semantics and the IR semantics return either a value (`ok`) or an explicit failure (`error`) when shape/type constraints are violated. Stating the result at this level preserves successful runs and well-defined failure behavior. The same invariant later lifts from the forward fragment to graph-level reasoning.

Theorem B.1: Forward lowering correctness

Let $\mathsf{lower}(p, \theta)$ be the IR obtained by lowering a forward program p with parameters θ, and let $\mathsf{Eval}_{\mathsf{src}}$ and $\mathsf{Eval}_{\mathsf{ir}}$ denote exception-valued evaluation of the source fragment and IR, respectively. Then for all inputs x,

$$\mathsf{Eval}_{\mathsf{ir}}(\mathsf{lower}(p, \theta), x) \;=\; \mathsf{Eval}_{\mathsf{src}}(p, \theta, x).$$

The following theorem targets the verifier/demo fragment. A single theorem for arbitrary user-written TORCHLEAN programs in the current higher-order/tagless-final embedding requires a logical-relations or parametricity development, or a different frontend encoding; that extension remains future work.

Theorem B.2: Eager–graph correspondence

For any fixed eager execution run that produces a computation tape T with forward values v_i and a backward pass that computes gradients g_i, there exists a well-typed SSA/DAG graph G such that:

- **Forward correspondence:** the eager forward values agree with the graph denotation ($v_i = [\![G]\!](x)_i$ for all nodes i).
- **Backward correspondence:** the eager backward gradients agree with graph backpropagation ($g_i = \mathrm{backpropCtx}(G, x, seed)_i$ for all nodes i).

B.4. TORCHLEAN programs, Modules, and API design

The core idea behind TORCHLEAN is that a neural network model is a backend-generic *program* built from a small, well-defined `Ops` interface. This avoids maintaining separate semantics for eager execution and compiled graphs: the same TORCHLEAN program can be interpreted by an eager backend that records a computation tape (as in PyTorch), or by a compiler backend that statically builds an SSA/DAG graph representation suitable for proofs and verification. Both paths share the same semantic meaning.

To make this concrete, consider a simple model consisting of a linear layer followed by a vector softmax, written as a backend-generic TORCHLEAN program:

```
def softmaxModel {a : Type} [Context a] [DecidableEq
     Shape] :
    TorchLean.Program a (paramShapes ++ [xShape])
     yShape :=
  fun {m} _ _ =>
    fun w b x => do
      let logits <- TorchLean.linear (m := m)
        (inDim := inDim) (outDim := outDim) w b x
      TorchLean.softmax (m := m) (s := yShape) logits
```

This definition is polymorphic in the scalar type α (which can be Float, IEEE32Exec, or $\mathbb{R}$) and in the backend monad m, which allows the same model to be executed in different contexts. The resulting term is structured enough to be compiled into our intermediate representation and fed directly to verification passes, reducing the gap between "the model as written" and "the model as verified."

For supervised learning workflows, we package models together with their training setup using a `ScalarModuleDef` structure. This bundles the initial parameters with a scalar loss function into a single object:

```
structure ScalarModuleDef (paramShapes inputShapes :
     List Shape) where
  initParams : Torch.TList Float paramShapes
  loss :
    forall {a : Type}, [Context a] -> [DecidableEq
     Shape] ->
      TorchLean.Program a (paramShapes ++ inputShapes)
     Shape.scalar
```

We store initial parameters as Float literals primarily for ergonomic reasons: it keeps small examples readable, allowing us to write `tensorND!` blocks without constantly fighting type casts. The cast from Float to the chosen scalar backend is treated as part of the *workflow layer* rather than part of the core semantics, which keeps the specification layer clean while maintaining flexibility.

Using this structure, a typical training loop is similar to PyTorch code. When instantiating a module, we choose both the scalar backend (via a `Float -> a` cast function) and the execution mode (`.eager` or `.compiled`):

```
open Runtime.Autograd.TorchLean.Module

let m <- defn.instantiate (cast := cast) (backend := .
     eager)
for _ in List.range T do
  let _ <- ScalarModule.forward m xs
  let _ <- ScalarModule.backward m xs
  ScalarModule.step m lr xs
```

For stateful optimizers (e.g. Adam), we can step using an explicit optimizer state aligned with the parameter shapes:

```
open Runtime.Autograd.TorchLean

let opt := Optim.adam (lr := lr) (beta1 := 0.9) (beta2
     := 0.999) (epsilon := 1e-8)
let st <- ScalarModule.initOptim m opt
for _ in List.range T do
  let st' <- ScalarModule.stepWith m opt st xs
  -- continue with st'
```

We introduce this wrapper to match the expected workflow (define a model, run a training loop) while keeping the resulting computation connected to the same SSA/DAG graph semantics that our proofs and verifiers consume.

Some workloads require *integer-valued indices* sourced from data (e.g., class labels for classification losses, or token/row indices for embedding lookups). Mainstream frameworks typically represent these indices as integer tensors that participate in the same runtime graph, even though gradients do not flow through the indices themselves. TORCHLEAN makes this separation explicit: the differentiable graph remains single-dtype over the chosen scalar domain, while indices are provided through a separate *non-differentiable* channel in the `Session` interface. In the `Session` interface, users supply scalar or batched indices as external inputs, and index-dependent operators (e.g., row-gather / embedding lookup) treat these indices as read-only selectors while gradients flow only through the floating-point tensor values (e.g., into the embedding matrix). This design avoids mixed-dtype graphs while still supporting standard ML patterns such as cross-entropy with integer labels and embedding-style table lookups.

What is `fderiv` / `HasFDerivAt`? In Lean/mathlib,

Table 4. Autograd operator coverage (Fréchet-derivative level). "Global" means we have a `NodeFDerivCorrect` proof (a `HasFDerivAt` theorem without side conditions). "Pointwise" means we prove `NodeFDerivCorrectAt` under explicit hypotheses that rule out non-differentiable points (e.g., no zeros/ties). Operators listed under "Not covered" may still be executable (and may have a deterministic backprop convention), but are not claimed correct at the level of classical derivatives.

OpKind / primitive	Global `HasFDerivAt`	Pointwise (side conditions)	Not covered
Linear / shape ops	`add`, `sub`, `mul_elem`, `linear`, `matmul`, `conv2d`, `broadcastTo`, `reshape`, `flatten`, `permute`, `reduce_sum`, `reduce_mean`, `sum`, `concat` (curated axes), `swap_first_two`, `transpose3d_last_two`	–	–
Smooth nonlinearities / losses	`tanh`, `sigmoid`, `exp`, `square`, `sinh`, `cosh`, `softplus`, `gelu` (tanh approximation), `silu`, `softmax`/`log_softmax` (curated axis), `layernorm` (curated axis), `mse_loss`, cross-entropy/NLL/BCE/KL pointwise losses, smooth surrogates `safe_log` and `smooth_abs` (with $\varepsilon > 0$)	–	–
Non-smooth / domain-sensitive	–	`relu`, `abs`, `log`, `inv`, `sqrt`, `max_elem`/`min_elem` (no ties)	–
Structured graph families	fixed-mask dropout, scaled dot-product attention, unmasked MHA, residual MHA, post-norm transformer sublayers, BatchNorm-like graphs, one-step Elman RNN, gather-row / embedding lookup adjointness	–	full GPT/ViT stacks, masked causal decoder blocks, full recurrent/state-space sequence theorems, FNO spectral-conv training paths
Nondifferentiable / workflow ops	–	–	`detach`, `rand_uniform`, stochastic masks beyond fixed-mask dropout, pooling (`max_pool2d`, `avg_pool2d`), executable CUDA/cuBLAS/cuFFT kernels

`fderiv` denotes the (Fréchet) derivative of a function between finite-dimensional real vector spaces, represented as a linear map; `HasFDerivAt` is the predicate that a function has a specified Fréchet derivative at a point. Informally, this is the standard Jacobian-level notion of differentiability: for $f : \mathbb{R}^n \to \mathbb{R}^m$, `fderiv` returns the Jacobian $Df(x)$ as a linear operator, and reverse-mode backpropagation computes its adjoint action on a cotangent seed. For non-smooth or domain-sensitive primitives (e.g., ReLU at 0, log at nonpositive inputs), we prove a pointwise variant that requires explicit side conditions ruling out those problematic points.

```
-- Sketch: indices are non-differentiable; gradients
     flow into emb only.
sess   := Session.new(...)
emb    := const(sess, W)          -- differentiable
     parameter tensor
idx    := input_indices(sess)     -- non-differentiable
     indices from data
rows   := gather_rows(sess, emb, idx)
loss   := ...
backward(loss)  -- dW accumulates; no gradients w.r.t.
     idx
```

Operator coverage across layers. TORCHLEAN exposes a shared set of primitives through the operator-tagged IR (`NN.IR.OpKind`) for both eager execution and proof-linked compiled execution. Autograd correctness is established for a curated subset of these primitives at the Fréchet-derivative level: Table 4 summarizes which ops are proved globally (`NodeFDerivCorrect`/`HasFDerivAt`), which are proved pointwise under explicit side conditions (`NodeFDerivCorrectAt` for non-smooth or domain-sensitive ops), and which are not covered by classical-derivative proofs (though they may remain executable under deterministic conventions). For verifier integration, the TORCHLEAN→IR compiler targets a conservative forward, first-order fragment, while the LiRPA stack provides IBP broadly and a basic CROWN-style affine pass with sound fallbacks when tighter relaxations are not implemented. Appendix B.6 gives the corresponding coverage details.

Control flow, variable-length sequences, indexing, and state. Our verified/compiled semantics targets SSA/DAG computation graphs: a *finite*, side-effect-free dataflow artifact evaluated in topological order. As a result, *data-dependent branching and loops* are not directly part of the verifier IR. Instead, dynamic control flow is handled by *reification* into a finite graph when the structure is known at compile time—for example, unrolling an RNN/GRU/LSTM cell for a fixed `seqLen`, or representing Transformers as acyclic attention blocks. This restriction matches the intended verification regime: certificates and proofs are stated about a fixed graph denotation $[\![G]\!]$, rather than about

an open-ended program whose control flow depends on runtime data.

Indexing requires special care because most indexing operators consume *integer-valued indices* (labels, token ids, embedding row selectors) that are inherently non-differentiable and introduce mixed-dtype graphs (float activations plus integer tensors). Mainstream ML runtimes permit such mixed graphs, but they complicate a semantic setting where we want a single, uniform scalar domain for the differentiable graph (to support clean denotations, autograd theorems, and bound propagation). TORCHLEAN therefore separates concerns: the differentiable graph remains single-dtype over the chosen scalar domain, while indices are supplied through an explicit *non-differentiable* channel in the `Session` API (e.g., scalar/batched naturals for labels and lookup indices). Index-dependent ops (e.g., embedding lookup / row gather) treat these indices as read-only selectors; gradients flow only through the floating-point values (e.g., into the embedding table or downstream layers), not through the indices themselves. On the verifier path, TORCHLEAN supports conservative indexing patterns that admit a clean lowering to the IR (e.g., scalar/row gathers reduced to one-hot selection and `matmul`); general integer-indexed `gather`/`scatter`, variable-shape slicing, and shape-changing data-dependent indexing remain outside the verified IR.

Finally, *state* is modeled explicitly rather than implicitly. The verifier semantics and most proofs assume layers are pure functions of their inputs (and any provided parameters), which is essential for treating the model as a mathematical object with a stable denotation. This is straightforward for affine layers and activations, but it requires care for layers with PyTorch-style internal updates (e.g., BatchNorm running mean/variance). In TORCHLEAN, such updates are performed outside the backend-generic op set: batch statistics can be computed from data, and running buffers updated explicitly in the imperative session layer. This design keeps the denotation used by verification free of hidden mutation, and makes any remaining state/update assumptions and trust boundaries visible at the API level.

B.5. Autograd verification

Training and verification pipelines routinely rely on gradients: optimization steps, sensitivity analyses, and derivative-based certificates (e.g., Lyapunov or PINN residual bounds) all assume that the computed derivatives match the model's intended semantics. In mainstream ML systems, backpropagation is trusted because it is heavily tested, but its correctness is rarely stated as a theorem about the exact computation being executed. TORCHLEAN makes this link explicit: we prove, for any well-typed SSA/DAG computation graph, that reverse-mode backpropagation computes the adjoint Fréchet derivative of the graph denotation. This turns gradient correctness into a *semantic property* of the same graph that is executed and verified, enabling derivative-dependent guarantees to compose cleanly with the rest of the framework rather than resting on an implicit trust assumption.

The proof is organized in two stages that separate algorithmic correctness from analytic interpretation. This structure makes the development modular and reusable across different scalar domains.

Stage 1: Algebraic adjointness (backend-generic). We first prove a graph-level adjointness law that does *not* depend on real analysis. The statement is parametric over an abstract scalar interface (our `Context` typeclass): any scalar domain that supports the required ring/ordering operations (e.g., reals, floats, intervals) can instantiate the theorem. At this level, reverse-mode backpropagation is characterized as computing a *vector–Jacobian product* (VJP) that is adjoint to the *Jacobian–vector product* (JVP) with respect to a dot product on *tensor contexts* (i.e., heterogeneous tuples of tensors matching the graph's typed input interface). Intuitively: pushing a perturbation forward (JVP) and pushing a cotangent backward (VJP) are dual operations, and their duality is captured by a single inner-product identity.

Contexts and dot products. Let $\langle \cdot, \cdot \rangle_\Gamma$ denote the dot product on the input context Γ (a typed tuple of tensors), defined by summing tensorwise dot products over all inputs/parameters. This dot product is bilinear and symmetric, which is exactly the structure needed for adjointness statements.

Forward/JVP/VJP as programs on a graph. For a well-typed SSA/DAG graph G and input x, we define three evaluation procedures: (i) forward evaluation $\mathrm{Eval}_G(x)$, (ii) forward-mode JVP $\mathrm{JVP}_G(x; \delta x)$, and (iii) reverse-mode backprop/VJP $\mathrm{VJP}_G(x; \bar{y})$. All three follow the graph's topological order (forward) and reverse-topological order (backward).

```
# Forward evaluation (topological order)
Eval_G(x):
  for node i in topo_order(G):
    v[i] = op_i(v[parents(i)])
  return v[out]

# Forward-mode JVP (propagate tangents)
JVP_G(x; dx):
  (v, _) = Eval_G(x)            # reuse forward values
  dv[input] = dx;  dv[others] = 0
  for node i in topo_order(G):
    dv[i] = J_op_i(v[parents(i)]) * dv[parents(i)]
  return dv[out]

# Reverse-mode VJP / backprop (propagate cotangents)
VJP_G(x; seed):
  (v, _) = Eval_G(x)
  bar[out] = seed; bar[others] = 0
  for node i in reverse(topo_order(G)):
    for p in parents(i):
```

```
        bar[p] += VJP_op_i(p, v[parents(i)], bar[i])
    return bar[input]              # gradients w.r.t.
      inputs/params
```

Theorem B.3: Graph adjointness (algebraic)

For any well-typed SSA/DAG graph G (over any `Context`) and any x, δx, and cotangent seed $\bar{y}$,

$$\langle \mathrm{JVP}_G(x;\delta x),\, \bar{y}\rangle \;=\; \langle \delta x,\, \mathrm{VJP}_G(x;\bar{y})\rangle.$$

Stage 2: Analytic upgrade (real calculus). Stage 1 establishes a purely algebraic adjointness law over an abstract scalar interface. In Stage 2 we specialize to $\mathbb{R}$ and connect that law to standard multivariate calculus. The key step is to relate shape-indexed tensor spaces to finite-dimensional Euclidean spaces: for each tensor shape s, let $n = \mathrm{size}(s)$ and identify tensors with vectors in $\mathbb{R}^n$ via a (total) vectorization map and its inverse. This lets us interpret graph evaluation as a function $\mathsf{EvalVec}_G : \mathbb{R}^{N_{\text{in}}} \to \mathbb{R}^{N_{\text{out}}}$ between Euclidean spaces and use the usual Fréchet derivative (Jacobian as a linear map).

Vectorization and inner products. Let $\mathrm{vec}_s : \mathrm{Tensor}(\mathbb{R}, s) \to \mathbb{R}^{\mathrm{size}(s)}$ be the flattening map (with inverse unvec_s). We define the tensor dot product $\langle a, b\rangle_s$ by summing coordinatewise products over the shape; the bridge theorem states that this dot product coincides with the standard Euclidean inner product after vectorization.

```
# Conceptual definition (shape-indexed)
vec_s   : Tensor(R,s) -> R^{size(s)}      # flatten
unvec_s : R^{size(s)} -> Tensor(R,s)      # reshape

# Bridge property
< a, b >_s  ==  < vec_s(a), vec_s(b) >_{R^{size(s)}}
```

Theorem B.4: Tensor–vector inner-product bridge

For any shape s and tensors $a, b \in \mathrm{Tensor}(\mathbb{R}, s)$,

$$\langle a, b\rangle_s \;=\; \big\langle \mathrm{vec}_s(a),\, \mathrm{vec}_s(b)\big\rangle_{\mathbb{R}^{\mathrm{size}(s)}}.$$

From local calculus to graph calculus. For each primitive operator (node kind), we prove a standard calculus fact: its forward-mode JVP coincides with applying its Fréchet derivative to a tangent vector at the current input (and similarly, its VJP coincides with applying the adjoint of that derivative to a cotangent seed). Combining these node-level facts with the SSA/DAG structure yields a global theorem for any well-typed graph. In finite dimensions, the adjoint of a linear map is just the transpose with respect to the Euclidean inner product; operationally, this is exactly what reverse-mode backprop computes.

Theorem B.5: Backpropagation computes the adjoint derivative

Let G be a well-typed SSA/DAG graph over $\mathbb{R}$ and let $\mathsf{EvalVec}_G$ denote its vectorized denotation. Assume the graph satisfies the local differentiability hypotheses for its primitives (i.e., `GraphFDerivCorrect` G). Then for any input x and cotangent seed $\bar{y}$,

$$\mathsf{VJP}_G(x;\bar{y}) \;=\; \big(D\,\mathsf{EvalVec}_G(x)\big)^{\top}\,\bar{y},$$

where $D\,\mathsf{EvalVec}_G(x)$ is the Fréchet derivative (Jacobian as a linear map) and $(\cdot)^{\top}$ denotes its adjoint (transpose) with respect to the standard inner product.

Interpretation. The theorem states that reverse-mode backprop is not merely an implementation heuristic: it computes the mathematically correct cotangent propagation associated with the derivative of the graph denotation. For non-smooth or domain-sensitive primitives (e.g., ReLU at 0, log at nonpositive inputs), we use a pointwise variant with explicit side conditions that rule out problematic points.

B.6. Node correctness and primitive operation proofs

The global backprop theorem is proved once for SSA/DAG graphs by composing *local* facts about each primitive. Accordingly, each primitive operator (node kind) must supply a correctness lemma connecting its implemented JVP/VJP rules to the mathematical derivative of its forward map. We use two variants: a *global* form for primitives that are smooth everywhere, and a *pointwise* form for non-smooth or domain-sensitive primitives (handled in the next paragraph).

Theorem B.6: Global node correctness (smooth primitives)

Let op be a primitive with forward map $f : \mathbb{R}^n \to \mathbb{R}^m$, and implemented tangent and cotangent rules $\mathrm{JVP}_{\mathrm{op}}(x;\delta x)$ and $\mathrm{VJP}_{\mathrm{op}}(x;\bar{y})$ (as used by the Stage 1 algorithms). We say op is *globally correct* if for every x there exists a (continuous) linear map $Df(x)$ such that

$$\mathrm{JVP}_{\mathrm{op}}(x;\delta x) \;=\; Df(x)\,\delta x \quad \text{and}$$

$$\mathrm{VJP}_{\mathrm{op}}(x;\bar{y}) \;=\; \big(Df(x)\big)^{\top}\bar{y}$$

for all tangents δx and cotangents $\bar{y}$. Equivalently, $\mathrm{JVP}_{\mathrm{op}}$ and $\mathrm{VJP}_{\mathrm{op}}$ satisfy the adjointness identity $\langle \mathrm{JVP}_{\mathrm{op}}(x;\delta x), \bar{y}\rangle \;=\; \langle \delta x, \mathrm{VJP}_{\mathrm{op}}(x;\bar{y})\rangle$ under the standard inner product.

This is the exact local hypothesis required by the Stage 2 upgrade: once each node kind used in a graph satisfies the above property, the previously stated graph theorem

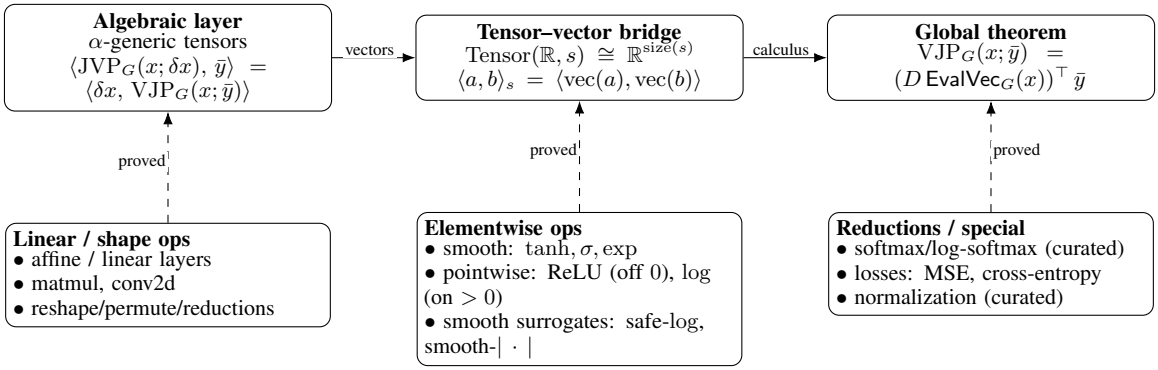

Figure 9. Two-stage autograd proof architecture with primitive coverage. Stage 1 proves adjointness between JVP_G and VJP_G over abstract scalars; Stage 2 bridges tensors to Euclidean spaces and upgrades to the Fréchet-derivative statement $\mathrm{VJP}_G(x;\bar{y}) = (D\,\mathsf{EvalVec}_G(x))^\top \bar{y}$. Bottom row summarizes operator classes covered by local derivative lemmas; see Table 4 for details.

(Theorem 2.1 / Theorem B.5) follows by SSA/DAG composition.

Example (affine layer). For $f(x) = Wx + b$, the derivative is the constant linear map $Df(x) = W$. Our implementation sets $\mathrm{JVP}_{\mathrm{op}}(x;\delta x) = W\delta x$ and $\mathrm{VJP}_{\mathrm{op}}(x;\bar{y}) = W^\top \bar{y}$, so the local theorem holds immediately. In Lean, the differentiability fact is discharged using the standard lemma that affine maps have constant Fréchet derivatives (e.g., `hasFDerivAt_affine`), after which the equalities above reduce to linear-algebra identities.

Pointwise correctness and kink points. For non-smooth or domain-sensitive primitives we use a pointwise predicate (our `NodeFDerivCorrectAt`): the same local condition as in Theorem B.6 holds, but only at a specific evaluation point x under explicit side conditions that guarantee classical differentiability (e.g., ReLU requires all relevant coordinates $\neq 0$; $\log$ requires inputs > 0; inv/div require nonzero denominators; $\max$ / $\min$ require no ties).

Theorem B.7: Pointwise node correctness (non-smooth primitives)

Let op be a primitive with forward map $f : \mathbb{R}^n \to \mathbb{R}^m$ and implemented rules $\mathrm{JVP}_{\mathrm{op}}(x;\delta x)$ and $\mathrm{VJP}_{\mathrm{op}}(x;\bar{y})$. Fix a point x and assume an explicit side condition $\mathcal{H}_{\mathrm{op}}(x)$ that guarantees classical differentiability at x (e.g., for ReLU: all relevant coordinates $\neq 0$; for $\log$: inputs > 0; for $\max$ / $\min$: no ties). Then there exists a linear map $Df(x)$ such that for all δx and $\bar{y}$,

$$\mathrm{JVP}_{\mathrm{op}}(x;\delta x) \;=\; Df(x)\,\delta x \qquad \text{and}$$

$$\mathrm{VJP}_{\mathrm{op}}(x;\bar{y}) \;=\; \big(Df(x)\big)^\top \bar{y}.$$

Equivalently,

$$\big\langle \mathrm{JVP}_{\mathrm{op}}(x;\delta x),\, \bar{y}\big\rangle \;=\; \big\langle \delta x,\, \mathrm{VJP}_{\mathrm{op}}(x;\bar{y})\big\rangle.$$

At kink points where the Fréchet derivative is undefined, we do *not* claim a calculus-level result; instead, the runtime still defines a deterministic executable backprop convention (as in mainstream autodiff systems) and we justify these rules at the *algebraic* level via the Stage 1 adjointness law, treating them as part of the internal semantics. Establishing that such conventions correspond to valid generalized derivatives (e.g., Clarke subgradients) at the kink points encountered during optimization is a separate, more delicate direction beyond Fréchet-derivative correctness.

Higher-order derivatives. Our proved autograd theorem is a first-derivative result: it characterizes reverse-mode as computing the adjoint Fréchet derivative of the forward denotation. We do *not* obtain second derivatives by recursively applying this theorem to the backward pass, since the backward graph can introduce non-smooth, branch-dependent primitives (e.g., ReLU gates) that complicate a clean calculus-level story. In the PINN case studies, we instead bound u' and u'' via derivative-aware bound propagation on the original operator-tagged forward graph (a directional first-derivative pass and a dedicated second-derivative pass for low-dimensional inputs over the smooth operator subset used in the demos), avoiding differentiation of the backward graph altogether. The runtime includes executable higher-order utilities (e.g., forward-over-reverse/HVPs via dual-number techniques), but full higher-order correctness requires extending the local node library and proof obligations to higher derivatives (including handling or excluding kinks), and then lifting the same SSA/DAG composition arguments to those higher-order rules.

Primitive operation proofs. Figure 9 summarizes the local derivative lemmas that feed the global SSA/DAG theorem. Rather than proving autograd correctness per model, we prove correctness *per primitive* and compose these facts along the graph. The proof work falls into three recurring patterns.

(1) Linear primitives (affine maps). For primitives whose forward maps are affine/linear (e.g., `linear`, `matmul`, `conv2d`), the derivative is the corresponding linear map, and the VJP is its adjoint (transpose), exactly matching standard backprop rules. Conv2D is handled by viewing convolution as a linear operator on flattened tensors and proving that our implemented backward rule coincides with the adjoint of that operator.

Theorem B.8: Example: affine node rule

For $f(x) = Wx + b$, we have $Df(x) = W$ and $\mathrm{VJP}(x; \bar{y}) = W^\top \bar{y}$.

(2) Elementwise primitives (coordinatewise calculus). For coordinatewise nonlinearities (e.g., $\tanh$, sigmoid, $\exp$), we lift scalar derivatives to tensors/vectors by showing that the derivative of the elementwise map is diagonal in the standard basis. For non-smooth or domain-sensitive elementwise ops (e.g., ReLU, $\log$, $\sqrt{\cdot}$, $1/x$), we use the pointwise variant away from kink/singularity points (as summarized in Table 4).

Theorem B.9: Example: elementwise lifting (informal)

If $f : \mathbb{R} \to \mathbb{R}$ is differentiable at each coordinate of $x \in \mathbb{R}^n$, then $D(f^{\odot})(x)$ is the diagonal linear map with entries $f'(x_i)$.

(3) Reductions and normalization (coupled coordinates). Operations like softmax/log-softmax and normalization layers couple coordinates via a shared denominator or statistics, so we prove their derivatives directly from their defining formulas and show that the resulting VJP matches the implemented backward rule. Standard losses (e.g., MSE, cross-entropy) then follow by composition: we treat them as maps from logits to a scalar and apply the chain rule on top of the network's derivative.

Theorem B.10: Example: softmax differential (informal)

Let $s = \mathrm{softmax}(x)$. Then $D\,\mathrm{softmax}(x)[dx] = s \odot \big(dx - \langle s, dx\rangle \mathbf{1}\big)$.

Together, these primitive proofs populate the local hypothesis used by the global SSA/DAG theorem; the detailed coverage matrix (global vs. pointwise) appears in Table 4.

B.7. Graph-level composition and the reverse-mode algorithm

This subsection explains how the *graph-level* reverse-mode theorem (Theorem 2.1) is obtained from the *local* node lemmas (Theorems B.6 and B.7) using SSA/DAG composition. The proof is modular: once each primitive node kind is equipped with a correct JVP/VJP rule, correctness lifts automatically to any well-typed graph built from those primitives.

Induction over SSA/DAG structure. A well-typed SSA/DAG graph admits a topological evaluation order, so its denotation $[\![G]\!]$ can be seen as a composition of node-level functions applied to previously computed values. The global proof proceeds by induction over the graph (equivalently, over the node list in SSA order): assume a prefix graph is correct, then extend it by one locally-correct node. The forward denotation of the extended graph is a composition, so its derivative follows by the chain rule, and the adjoint derivative follows by reversing the order of composition. This is exactly what reverse-mode implements.

Reverse-mode algorithm (VJP form). To connect the theorem statement to familiar autodiff intuition, we write the backpropagation procedure as a cotangent propagation on the DAG. The symbol *seed* (often written $\bar{y}$) denotes the cotangent supplied at the graph output: it specifies which linear objective of the output we are differentiating. For scalar losses, the seed is typically 1; for vector outputs (e.g. logits), the seed can be any cotangent vector and yields the corresponding vector–Jacobian product.

```
# Forward pass (topological order)
Eval_G(x):
  for node i in topo_order(G):
    v[i] = op_i(v[parents(i)])
  return v, v[out]

# Reverse pass (cotangent propagation)
VJP_G(x; seed):
  (v, _) = Eval_G(x)
  bar[out] = seed;  bar[others] = 0
  for node i in reverse(topo_order(G)):
    for p in parents(i):
      bar[p] += VJP_op_i(p, v[parents(i)], bar[i])
  return bar[input]      # gradients for inputs/params
```

Why accumulation is correct on DAGs. If a node value is used by multiple downstream nodes, it receives multiple cotangent contributions in the reverse sweep. The algorithm therefore *adds* these contributions (the += line). This is not an implementation detail: it is the mechanism that makes reverse-mode correct for graphs with shared subexpressions. Algebraically, correctness follows from the adjointness identity in Stage 1 (Theorem B.3): the dot product is bilinear, so contributions from different paths combine by summation.

From VJP to gradients (the scalar-loss special case). Many presentations restrict to scalar losses $L : \mathbb{R}^n \to \mathbb{R}$ and identify reverse-mode with $\nabla L(x)$. Our statement is more general: for $f : \mathbb{R}^n \to \mathbb{R}^m$ and a cotangent seed $\bar{y} \in \mathbb{R}^m$, reverse-mode returns the adjoint derivative applied to the seed, $(Df(x))^\top \bar{y}$, i.e. a vector–Jacobian product (Theorem 2.1). The scalar-loss gradient is recovered

by taking $m = 1$ and $\bar{y} = 1$ (or by composing a vector-valued network with a scalar loss and applying the chain rule). This formulation is essential for ML workloads where intermediate quantities are tensor-valued (e.g. logits, features, attention blocks), but training and verification often require derivatives of *particular* scalar objectives derived from them.

Parameter handling (typed parameter packs). Modern models have heterogeneous parameter packs: a linear layer carries a matrix and a bias vector, a Conv2D layer carries a 4D kernel and bias, and attention blocks carry multiple projection matrices. TORCHLEAN represents such packs as a *shape-indexed heterogeneous list* (a typed product) rather than a string-keyed map. A parameter pack is typed by a list of shapes $[s_1, \ldots, s_k]$ and contains one tensor of each corresponding shape; semantically this is the finite product space $\mathrm{Tensor}(\alpha, s_1) \times \cdots \times \mathrm{Tensor}(\alpha, s_k)$. This choice makes parameters part of the same typed interface as ordinary graph inputs: compilation treats them as additional inputs to the SSA/DAG graph, and reverse-mode returns cotangents for *both* data inputs and parameters in the same structured form. Optimizers then become shape-preserving transformations on parameter packs (e.g., a pointwise update that zips parameters with their gradients), so a mismatch between a parameter tensor and its gradient is ruled out by the typechecker rather than discovered at runtime. By contrast, common ML runtimes store parameters in mutable, string-keyed containers for convenience, which pushes the shape contract into informal conventions; in TORCHLEAN, the shape contract is intrinsic, and entire classes of "wrong tensor paired with wrong gradient" bugs become unrepresentable.

B.8. Core Definitions and IR Semantics

We focus on the core definitions that the rest of TORCHLEAN builds on: (i) the *shape-indexed* tensor semantics used as the reference meaning for proofs and verification, (ii) the common scalar interface, and (iii) the operator-tagged SSA/DAG IR. We include these excerpts because extending TORCHLEAN (adding ops, adding backends, or adding verification rules) requires understanding the semantic core these components share.

Shape-indexed tensors (why this representation). The most basic design decision is to make tensor shapes part of the *type*. In mainstream ML systems, shapes are runtime values and shape mismatches appear late (as runtime errors or, worse, silent broadcasting mistakes). In contrast, we want the shape contract to be an invariant of the logic: if an expression typechecks, it *cannot* be ill-shaped. This yields cleaner theorem statements (no repeated "shape matches" premises) and makes the semantic core robust to conventions that are easy to get wrong (layout, reshaping discipline, implicit broadcasting).

Why functional tensors rather than flat arrays? For the semantic layer, we prioritize *proof ergonomics* and *semantic clarity*. We represent a tensor structurally as a total indexing function: a tensor of shape `dim n s` is literally a function `Fin n -> Tensor a s`. This makes theorems about tensor programs follow the same recursion as the datatype. For example, commutativity of elementwise addition is proved by induction on the shape: the scalar case is immediate, and the `dim` case reduces to the induction hypothesis pointwise. With a flat array representation, such proofs require explicit index arithmetic and bounds reasoning, which is significantly harder to automate and more brittle as the operator library grows. Finally, the spec layer intentionally does *not* commit to a concrete storage layout (row-major vs. column-major), so it remains a stable reference meaning across execution backends.

Listing 5. Core shape-indexed tensor definitions (semantic layer).

```
namespace Spec

/-- Shapes: scalar, or "n copies of a
    subshape". -/
inductive Shape where
  | scalar : Shape
  | dim    : Nat -> Shape -> Shape

/-- Shape-indexed tensors, represented
    structurally. -/
inductive Tensor (a : Type) : Shape -> Type
     where
  | scalar : a -> Tensor a .scalar
  | dim    : forall {n s}, (Fin n -> Tensor
     a s) -> Tensor a (.dim n s)
```

How to read this definition. The constructor `Tensor.scalar` is a scalar value. The constructor `Tensor.dim` stores an index function that returns the i-th subtensor. Thus:

$$\mathrm{Tensor}(\alpha, \mathtt{dim}\, n\, s) \equiv (\mathtt{Fin}\, n \to \mathrm{Tensor}(\alpha, s)),$$

i.e., a length-n vector of subtensors of shape s. This immediately gives canonical encodings of common ML shapes: a vector of length n is `dim n scalar`; a matrix $m \times n$ is `dim m (dim n scalar)`; and a batch of images (batch size B, channels C, height H, width W) is `dim B (dim C (dim H (dim W scalar)))`.

Why this pays off in proofs. Most tensor operations are defined by recursion on shape, so proofs follow the same structure. For instance, elementwise addition on `dim n s` tensors is defined pointwise on the `Fin n` index and then recursively on s; extensional equality reduces tensor equality to pointwise equality of index functions. This makes basic algebraic properties (associativity, commutativity, distributivity) and shape-preservation properties easy to state and to prove.

Efficiency note (bridging to array-backed execution). The functional tensor representation is chosen for the *semantic reference* used by proofs and verification; it is not intended as the fastest execution format. For executable workflows, TORCHLEAN additionally provides a materialized, array-backed representation and a semantics-preserving bridge between the two, so programs can run efficiently while still referring to the same underlying meaning. We discuss this representation, its compilation path, and the resulting performance trade-offs in Section C (and use it throughout the runtime experiments).

Scalar polymorphism (one model, many semantics). Most specification-level definitions are polymorphic in the scalar type a, so the *same* model code can be interpreted over multiple numeric domains: $\mathbb{R}$ for proof level reference semantics, executable floating-point domains for runtime demos, and abstract/rounded domains for sound bounds and error envelopes. To make this practical, TORCHLEAN collects the numeric structure required by neural-network operators into a single typeclass, `Context a`. Intuitively, `Context a` is "the interface a scalar type must implement to run NN code."

Listing 6. Scalar interface for NN code (`Context`).

```
class Context (a : Type) extends
  Add a, Sub a, Mul a, Div a, Neg a,
  LT a, LE a, One a, Zero a,
  MathFunctions a, Numbers a where
  decidable_gt : DecidableRel (fun x y : a
    => x > y)
```

This interface includes: (i) ring-like arithmetic (`Add/Sub/Mul/Div/Neg`) used throughout linear algebra and losses; (ii) order structure (`LT/LE` and decidability) needed for conditionals and piecewise primitives (e.g., ReLU, `max/min`); (iii) constants (`Zero/One`) and numeric literals (`Numbers`); and (iv) transcendental functions (`MathFunctions`, e.g., `exp`, `log`, `tanh`, `sqrt`, `sin`, `cos`). By making the required scalar operations explicit, we can state and reuse theorems without committing to a particular numeric representation: a lemma proved for `Context a` specializes uniformly to $\mathbb{R}$, executable float models, and rounding/interval domains used by verification.

Graph IR structure (why SSA/DAG). Our verification substrate is an operator-tagged SSA/DAG intermediate representation (IR) with explicit node kinds and output shapes. The IR is intentionally simple: it is the *single semantic target* reused across execution, differentiation, and verification. More precisely, the IR is (i) *operator-tagged* so each node carries an explicit primitive identifier (an opcode) and any parameters needed to interpret it, and (ii) in *SSA/DAG form* so each intermediate value is defined exactly once and dataflow is acyclic.

Why a DAG? Most verifier pipelines (IBP/LiRPA/CROWN) operate on feedforward computation graphs: sound bounds are propagated along edges, which requires a well-defined evaluation order. A DAG gives exactly this: nodes admit a topological order, so evaluation and bound propagation are deterministic and total. This also makes proofs modular: semantic properties can be established by induction over the node list (SSA order), and verification passes can be defined as simple forward (or forward+backward) sweeps over the same graph object. Cycles and implicit control flow, by contrast, require separate fixpoint semantics (and additional invariants) for both execution and verification; TORCHLEAN handles such behavior by *reification* into finite graphs when needed (e.g., unrolling a fixed-length recurrent cell).

Why SSA? SSA (static single assignment) ensures every intermediate value has a unique definition. This simplifies both implementation and reasoning: backpropagation and certificate checking can attach metadata (values, bounds, dual variables) to node IDs without ambiguity, and gradient contributions from multiple consumers are accumulated by summation in the reverse sweep. SSA therefore makes "what does this gradient/bound refer to?" a structural property of the IR.

Core definition. Each node records: (i) an operation kind, (ii) the IDs of its parent nodes (its inputs), and (iii) a declared output shape. OpKinds carry any parameters required for interpretation (e.g., `conv2d` stores channel counts, kernel size, stride, padding; `softmax` stores an axis).

Listing 7. Operator tagged SSA/DAG IR (core excerpt).

```
inductive OpKind where
  | input
  | linear
  | relu
  | conv2d (inC outC kH kW stride padding :
    Nat)
  | softmax (axis : Nat)
  | layernorm (axis : Nat)
  | mse_loss

structure Node where
  id       : Nat
  parents  : List Nat
  kind     : OpKind
  outShape : Spec.Shape
```

Typing and denotation. Well-typedness checks that each node's parent shapes match what its `OpKind` expects and that `outShape` matches the primitive's output contract. The denotation $[\![G]\!]$ evaluates nodes in topological order (which is well-defined because the graph is acyclic) and is total on well-typed graphs. This denotational semantics is the object used by proofs (e.g., autograd correctness) and by verification passes (e.g., IBP/CROWN transfer rules):

Table 5. High-level comparison between PyTorch (Paszke et al., 2019) and TORCHLEAN.

Aspect	PyTorch	TORCHLEAN
Shapes	Dynamic runtime shapes; many errors are runtime exceptions.	**Shapes are part of types** (`Tensor` α `s`); many mismatches are untypeable.
Dtypes	Many numeric and integer dtypes; mixed-dtype graphs are common.	**Scalar-polymorphic** single-dtype graphs (one α per run); integer indices handled via a separate non-differentiable channel (`NatRef`/`NatVecRef`) in sessions.
Execution modes	Eager by default, with compilation/export toolchains (TorchScript, ONNX, AOT).	**Eager tape backend** and **proof-linked compiled** SSA/DAG backend share one semantics; the compiled graph is the verifier target by construction.
Autograd status	Widely tested and trusted, but not formally proved correct.	**Reverse-mode correctness theorem** for well-typed SSA/DAG graphs; eager runs linked to proved graphs.
Indexing	Tensor-valued indexing/slicing/gather/scatter across dtypes (e.g. `LongTensor`).	Typed indexing primitives and session-level Nat channels for labels/indices; not yet PyTorch-complete for tensor-valued integer indices.
Breadth/performance	Very broad op surface and ecosystem; highly optimized CPU/GPU kernels.	Curated op surface focused on verification-relevant primitives; extensible via a "new op" workflow; Lean execution prioritizes clarity/verification over performance.

bounds and certificates are interpreted against *the same* primitive meanings that execution uses.

Losses Much of the machine learning literature phrases training in terms of a scalar loss $L(\theta)$ and its gradient $\nabla_\theta L$, but in a tensor-typed setting, we found it important to make the "scalar" part explicit: losses are tensor programs followed by reductions. This design choice makes the structure of losses clear and enables precise reasoning about gradients. In the library, losses are ordinary TORCHLEAN programs that compute a tensor-valued quantity and then reduce it to a scalar. We provide both a primitive scalar loss (`mse_loss`) and a small `Loss` helper layer that mirrors common PyTorch conventions: MSE, cross-entropy/negative-log-likelihood variants (one-hot targets and index-based targets), and binary cross-entropy (including a stable "with logits" form), each with explicit reduction (`mean` or `sum`).

Extending TORCHLEAN across the stack. Adding a new op is a predictable, checkable workflow rather than an ad hoc engineering task. When we add a primitive to TORCHLEAN, we treat it as a complete commitment: it should have a spec meaning, a typing rule, and the transfer rules needed by theorems/verification. The workflow has five steps: (1) **Spec semantics:** Define the operation as a total function on shape-indexed tensors in the spec layer, with an explicit shape contract. If the operation is non-smooth or domain-sensitive (e.g. log, division, max), either define a safe/smoothed variant intended for theorem statements, or adopt the pointwise hypothesis style and document the required preconditions. (2) **IR support:** Add an op tag to the IR (`OpKind`) and define its typing rule (input and output shapes). This is what makes compilation and verification passes recognize the op uniformly. (3) **Autograd over $\mathbb{R}$:** Provide local JVP/VJP rules and prove the local adjointness law (or the pointwise variant). Once this lemma exists, the global theorem applies to any graph using the op. (4) **Verification transfer rules:** For IBP, define a sound transfer function on boxes. For affine relaxations, either implement and prove the relaxation, or import bounds as certificates and check the certificate constraints in Lean. (5) **Numeric backends:** Decide which execution backends support the op (Float, IEEE32Exec), and (when relevant) add local rounding/error lemmas so the op participates in graph level NF bounds.

B.9. TORCHLEAN vs. PyTorch

This comparison focuses on goals. PyTorch is an industrial execution framework optimized for throughput, hardware utilization, and ecosystem breadth; TORCHLEAN is a semantic interface whose main objective is to make training code and verifier-time artifacts coincide so that guarantees are stated about the *executed* artifact. Accordingly, TORCHLEAN does not attempt to compete with PyTorch on industrial-scale throughput; it prioritizes a precise semantic

link between user programs, IR graphs, runtime artifacts, and proofs. Optional CUDA kernels and fast runtime paths support larger examples, but the key contribution is the semantic substrate: performance-oriented backends can be added behind explicit contracts without changing what is being verified. TORCHLEAN is also not "just an API": the design choices directly enable the verification story. Compilation is a first-class path that lowers programs to a well-typed operator-tagged SSA/DAG IR, and eager execution records a tape that we prove corresponds to an equivalent well-typed IR graph (Theorem B.2). Once a graph exists, whether it was obtained eagerly or via lowering, the same semantic object is consumed by autograd theorems, bound-propagation passes, and certificate checkers.

C. Modern workflow theory and semantics: CUDA, attention, RL, diffusion, probability, SSL, and model families

In this section, we spell out the contracts behind the workflow examples used in the paper. For each family, we name the Lean object, write the equation or recurrence it denotes, and state what Lean checks versus what comes from a runtime or external producer. Concretely, attention is a masked softmax equation, an RL rollout is a finite recurrence, diffusion is a schedule plus a sampler step, and self-supervised learning is represented by a finite objective over masks or views.

C.1. CUDA/native FFI and runtime contracts

The CUDA path is an execution feature, not a second proof semantics. Lean's kernel checks theorem terms and definitions written in Lean; it does not inspect CUDA source code, GPU machine code, cuBLAS, cuFFT, driver behavior, memory allocation, or the order in which floating-point reductions are scheduled. In TORCHLEAN, a native kernel is therefore attached to a Lean operator whose denotation is already defined at the specification or IR level. The formal object remains the graph denotation; the native call is an optional implementation path.

For an operator tag τ with specification

$$[\![\tau]\!]_\alpha : \mathrm{Tensor}_\alpha(s_1) \times \cdots \times \mathrm{Tensor}_\alpha(s_k) \to \mathrm{Tensor}_\alpha(s),$$

a CUDA-backed implementation is treated through a conformance statement of the form

$$\mathsf{cuda}_\tau(x_1, \ldots, x_k) \approx [\![\tau]\!]_{\texttt{Float32}}(x_1, \ldots, x_k),$$

under explicit preconditions about dtype, shape, layout, device memory, determinism, and precision mode. When this relation is not proved in Lean, it is recorded as a runtime assumption and tested by conformance/regression checks. Thus a theorem about $[\![G]\!]$ does not become a theorem about arbitrary GPU execution unless the needed conformance hypothesis is supplied.

The implementation uses Lean's FFI discipline: Lean declarations expose an opaque runtime handle, while C/CUDA wrappers allocate device buffers, launch kernels, and copy results across the host/device boundary. The following schematic code shows the shape of the interface; the real implementation fixes the buffer layout and performs the corresponding runtime checks.

Listing 8. Schematic Lean-side boundary for a native CUDA primitive.

```
namespace Runtime.Autograd.Cuda

opaque Buffer : Type

@[extern "torchlean_cuda_bmm"]
opaque Buffer.bmm : Buffer -> Buffer -> IO
    Buffer

/-- Runtime-side check before calling the
    native kernel. -/
def bmmChecked (a b : TensorMeta) (ha :
    Buffer) (hb : Buffer) : IO Buffer := do
  checkMatmulShapes a.shape b.shape
  checkDType a.dtype .float32
  checkDType b.dtype .float32
  Buffer.bmm ha hb

end Runtime.Autograd.Cuda
```

Listing 9. Schematic C/CUDA wrapper; memory safety and launch behavior are outside Lean's kernel.

```
extern "C" LeanObj torchlean_cuda_bmm(LeanObj a,
    LeanObj b) {
  CudaBuffer* A = unwrap_buffer(a);
  CudaBuffer* B = unwrap_buffer(b);
  CudaBuffer* C = allocate_like_matmul(A, B);
  cublasSgemmStridedBatched(/* handles, shapes, strides,
     A, B, C */);
  return wrap_buffer(C);
}
```

This pattern is used for dense and batched matrix products, reductions, broadcasting, tensor views, convolution/pooling and transposed-convolution kernels, FFT/spectral operators, softmax/log-softmax, normalization kernels, attention-oriented and fused-attention paths, gather/scatter utilities, positional/RoPE helpers, selective-scan-style sequence kernels, and other runtime paths used by the examples. Several backward and reduction kernels use floating-point accumulation. Because floating-point addition is not associative, atomic accumulation can be mathematically standard while still not being bit deterministic across schedules. TORCHLEAN therefore exposes deterministic-reduction modes for selected paths and treats remaining scheduling choices as part of the deployment boundary.

The same policy applies to fused attention. The Lean specification can state and prove that a formal FlashAtten-

tion operator denotes standard masked scaled-dot-product attention. That equality is a statement about the mathematical operator in Lean. It is not a verification of a particular CUDA implementation, tiled online-softmax schedule, HBM/SRAM traffic pattern, or GPU kernel binary.

C.2. FlashAttention, masks, and fused-attention denotation

The attention specification is stated once, following the standard scaled-dot-product attention abstraction and the FlashAttention denotational contract used in the codebase (Dao et al., 2022). For query, key, and value tensors $Q \in \alpha^{n_q \times d}$, $K \in \alpha^{n_k \times d}$, $V \in \alpha^{n_k \times d_v}$, and an optional Boolean mask M, standard scaled dot-product attention is the relation

$$\mathrm{SDPA}(Q, K, V, M) = \mathrm{softmax}_M\left(\frac{QK^\top}{\sqrt{d}}\right)V. \quad (1)$$

The mask semantics are part of the specification. In the formal definition, masked entries contribute zero probability after the row-wise masked-softmax step; causal masks are therefore ordinary finite predicates over query/key positions, not a convention hidden inside a backend call.

FlashAttention is modeled as a fused route to the same denotation. A configuration record stores scheduling metadata such as query/key tile sizes, but these fields are intentionally ignored by the mathematical denotation: tiling changes how a backend may compute the row, not what function the row denotes. At the spec level, TORCHLEAN names an online/tiled operator and proves the central equality

$$\mathrm{onlineSoftmaxTiledAttention}(cfg, ctx) = \mathrm{scaledDotProductAttention}(ctx) \quad (2)$$

It also exposes the fused forward operator theorem

$$\mathrm{flashAttention}(cfg, ctx) = \mathrm{scaledDotProductAttention}(ctx) \quad (3)$$

and the corresponding VJP/backward contract

$$\mathrm{flashAttentionBackward}(cfg, ctx, \bar{y}) = \mathrm{scaledDotProductAttentionBackward}(ctx, \bar{y}). \quad (4)$$

These equalities are the formal content behind the paper's statement that "FlashAttention equals ordinary attention" in TORCHLEAN: the claim is a denotational equality between a fused specification operator and the standard masked-attention specification. It is deliberately not a proof that a particular CUDA source file implements Dao-style IO-aware tiling correctly. The native CUDA path is regression-tested against the composed attention path and is documented as an FFI trust boundary; the theorem surface stays inside the specification layer.

This split is useful for compiler style reasoning. A graph rewrite may replace a composed $QK^\top$-mask-softmax-PV subgraph by a fused attention operator only if both sides share the same typed attention context and mask convention. The theorem above is exactly the small equality such a rewrite needs.

Listing 10. Schematic attention contract: the fused operator is checked against masked SDPA.

```
structure AttentionCtx (a : Type) where
  Q K V : Tensor a _
  mask  : Option (Fin _ -> Fin _ -> Bool)
  scale : a

def sdpa (ctx : AttentionCtx a) : Tensor a
    _ :=
  maskedSoftmax (ctx.scale * (ctx.Q.matmul
    ctx.K.transpose)) ctx.mask
    |>.matmul ctx.V

def flashSpec (cfg : FlashCfg) (ctx :
    AttentionCtx a) : Tensor a _ :=
  -- The schedule fields in cfg affect the
    implementation, not the denotation.
  sdpa ctx

theorem flashSpec_eq_sdpa (cfg : FlashCfg)
    (ctx : AttentionCtx a) :
    flashSpec cfg ctx = sdpa ctx := rfl
```

C.3. Reinforcement-learning semantics: returns, Bellman operators, MDPs, and GridWorld

RL adds an environment boundary to the usual model/runtime boundary, so the appendix follows the standard discounted-dynamic-programming vocabulary of Bellman operators, MDPs, returns, and generalized advantage estimation (Bellman, 1957; Puterman, 1994; Sutton & Barto, 2018; Schulman et al., 2016). TORCHLEAN separates three levels: (i) pure rollout algebra over lists and finite tensors, (ii) Lean-based MDPs and environments, and (iii) external simulators that emit trajectories through a checked artifact boundary. Runtime collectors, optimizers, replay buffers, logging, and optional CUDA execution live outside this pure layer.

Rollout algebra. For a terminal flag d, define the continuation mask

$$m(d) = \begin{cases} 0, & d = \texttt{true}, \\ 1, & d = \texttt{false}. \end{cases} \quad (5)$$

The one-step backup, TD target, and TD residual are

$$B_\gamma(r, b, d) = r + \gamma\, m(d)\, b, \quad (6)$$

$$y_t = r_t + \gamma\, m(d_t) V(s_{t+1}), \quad (7)$$

$$\delta_t = y_t - V(s_t). \quad (8)$$

For a reward list $r_0, \ldots, r_{T-1}$, `discountedReturnsFrom` is the right-fold recurrence

$$G_T = b, \qquad G_t = r_t + \gamma G_{t+1}. \tag{9}$$

With done flags, the recurrence becomes

$$G_t = r_t + \gamma m(d_t) G_{t+1}, \tag{10}$$

so terminal episodes reset the bootstrap. Generalized advantage estimation is formalized by the reverse recurrence

$$A_t = \delta_t + \gamma \lambda m(d_t) A_{t+1}, \tag{11}$$

where each `AdvantageStep` stores r_t, $V(s_t)$, $V(s_{t+1})$, and d_t. The corresponding return target is recovered by $R_t = A_t + V(s_t)$. These definitions are intentionally over lists rather than fixed-shape tensors, because real rollouts have variable length due to termination/truncation; fixed-horizon tensor variants can be built on top when needed.

Deterministic finite MDPs. The simplest MDP layer uses finite state and action types $S = \mathrm{Fin}(n)$ and $A = \mathrm{Fin}(m)$. A deterministic finite MDP consists of an initial state, a total transition/reward/termination function

$$\mathrm{step} : S \times A \to S \times \mathbb{R} \times \{\texttt{terminated}, \texttt{truncated}\}, \tag{12}$$

and a discount factor. For a value table $v : S \to \alpha$, the state-action value induced by one step is

$$Q_v(s,a) = r(s,a) + \gamma m(d(s,a)) v(s'(s,a)). \tag{13}$$

The policy Bellman operator and optimality operator are

$$(T^\pi v)(s) = Q_v(s, \pi(s)), \tag{14}$$

$$(T^* v)(s) = \max_{a \in A} Q_v(s,a), \tag{15}$$

where the finite maximum is implemented with `Finset.sup'` over a nonempty finite action space. Value functions and action-value tables are typed tensors, so these operators can be used alongside the same tensor infrastructure as ordinary neural models.

Finite stochastic MDPs. The finite stochastic layer replaces the deterministic successor with a typed transition row $P(\cdot \mid s,a)$. A validity record requires nonnegative entries, rows summing to one, and $0 \le \gamma < 1$. The expected next value and Bellman action value are

$$\mathbb{E}[v(s') \mid s,a] = \sum_{s' \in S} P(s' \mid s,a) v(s'), \tag{16}$$

$$Q_v(s,a) = r(s,a) + \gamma m(d(s,a)) \sum_{s' \in S} P(s' \mid s,a) v(s'). \tag{17}$$

This layer gives the proof and runtime code a finite tensor representation for stochastic dynamics while avoiding the full measure-theoretic overhead when the state space is finite.

Listing 11. Schematic finite MDP and Bellman operator.

```
structure FiniteMDP (S A : Type) where
  reward : S -> A -> Float
  trans  : S -> A -> S -> Float       -- row
    -stochastic transition table
  done   : S -> A -> Bool
  gamma  : Float

def expectedNext (M : FiniteMDP S A) (v : S
     -> Float) (s : S) (a : A) : Float :=
  Finset.univ.sum (fun s' => M.trans s a s'
     * v s')

def bellmanPi (M : FiniteMDP S A) (pi : S
    -> A) (v : S -> Float) (s : S) : Float
    :=
  let a := pi s
  M.reward s a + M.gamma * contMask (M.done
     s a) * expectedNext M v s a
```

Measure-theoretic Markov-kernel MDPs. For continuous or general measurable spaces, TORCHLEAN defines an MDP with measurable state space S, action space A, and a mathlib Markov kernel $\kappa : S \times A \rightsquigarrow S$. The expected next value is the integral

$$\mathbb{E}[v(s') \mid s,a] = \int_S v(s')\, d\kappa(s,a)(s'), \tag{18}$$

and the Bellman policy operator is

$$(T^\pi v)(s) = r(s,\pi(s)) + \gamma m(d(s,\pi(s))) \int_S v(s')\, d\kappa(s,\pi(s))(s'). \tag{19}$$

The validity assumptions require κ to be a Markov kernel, reward and termination to be measurable, and $0 \le \gamma < 1$. The corresponding proof layer states standard discounted Bellman facts for bounded value functions, including monotonicity and contraction in the sup metric:

$$\|T^\pi v - T^\pi w\|_\infty \le \gamma \|v - w\|_\infty, \tag{20}$$

$$\|T^* v - T^* w\|_\infty \le \gamma \|v - w\|_\infty. \tag{21}$$

Because $\gamma < 1$, these contraction theorems support uniqueness of the fixed point for the Bellman operator under the stated metric assumptions.

GridWorld as a Lean-based environment. The concrete GridWorld uses coordinate states $(\mathrm{row}, \mathrm{col})$, four actions (up, down, left, right), border-clamped transitions, a goal cell, and a discount factor. Reaching the goal returns reward 0 and terminates; otherwise a transition receives reward -1. The environment has three views: a pure `Env`

view with explicit latent state, a deterministic finite MDP view obtained by flattening $(\mathrm{row}, \mathrm{col})$ to $\mathrm{Fin}(h \cdot w)$, and a finite stochastic view whose transition rows are one-hot. This gives the paper a small RL case where environment semantics, Bellman operators, value tables, and rollout summaries all live inside Lean.

External simulators. For CartPole-, Pong-RAM-, or other Gymnasium-style workflows, the simulator is not hidden inside a theorem. Python or another runtime produces observations, actions, rewards, log probabilities, and done/truncation flags. Lean checks the rollout schema and then consumes the artifact for losses or graph properties. Thus the verified object is a policy/value graph property, a Bellman/return calculation, or a checked rollout contract; the external simulator remains a named producer.

C.4. Diffusion, sampling, probability-flow ODEs, and sampler stability

Diffusion models exercise a different kind of semantics from classifiers, so we separate deterministic sampler equations from probability-law statements in the style of DDPM, DDIM, and score/probability-flow formulations (Ho et al., 2020; Song et al., 2021a;b). The objects of interest are schedules, noising maps, denoisers, reverse transitions, samplers, and sometimes probability laws. TORCHLEAN keeps these as scalar-polymorphic specification definitions so they can be evaluated with `Float`, executable finite-precision models, intervals, or $\mathbb{R}$, depending on the theorem or experiment.

Discrete VP schedules and forward noising. A discrete variance-preserving schedule of length T stores per-step variances β_t. The specification defines

$$\alpha_t = 1 - \beta_t, \tag{22}$$

$$\bar{\alpha}_0 = 1, \tag{23}$$

$$\bar{\alpha}_{t+1} = \bar{\alpha}_t \alpha_t. \tag{24}$$

The forward noising map `qSample` is the pure, total tensor function

$$q_t(x_0, \varepsilon) = \sqrt{\bar{\alpha}_t}\, x_0 + \sqrt{1 - \bar{\alpha}_t}\, \varepsilon. \tag{25}$$

At this spec level ε is an explicit tensor input. The probabilistic interpretation $\varepsilon \sim \mathcal{N}(0, I)$ belongs either to the probability theory layer or to the runtime sampler that produces noise.

Listing 12. Schematic diffusion schedule and forward noising map.

```
structure VPSchedule where
  beta     : Nat -> Float
  alphaBar : Nat -> Float

def qSample (sch : VPSchedule) (t : Nat)
    (x0 eps : Tensor Float s) : Tensor
    Float s :=
  let c0 := sqrt (sch.alphaBar t)
  let c1 := sqrt (1.0 - sch.alphaBar t)
  Tensor.map2 (fun x e => c0 * x + c1 * e)
    x0 eps
```

Epsilon prediction and DDPM loss. A denoising model is represented by an epsilon-prediction function

$$\varepsilon_\theta : (x, t) \mapsto \widehat{\varepsilon}. \tag{26}$$

The standard epsilon-prediction training loss is a named wrapper around mean squared error:

$$\mathcal{L}_\varepsilon(\theta; x_0, t, \varepsilon) = \|\varepsilon_\theta(q_t(x_0, \varepsilon), t/T) - \varepsilon\|_2^2. \tag{27}$$

This does not require a special loss semantics: it reuses the existing tensor MSE definition but gives the diffusion convention a domain-specific name.

DDPM reverse steps. The epsilon-parameterized reconstruction is

$$\widehat{x}_0(x_t, t) = \frac{x_t - \sqrt{1 - \bar{\alpha}_t}\, \varepsilon_\theta(x_t, t/T)}{\sqrt{\bar{\alpha}_t}}, \tag{28}$$

with totalized safe division and nonnegative square-root wrappers in the executable definition. The reverse DDPM step $x_t \mapsto x_{t-1}$ uses

$$\mu_t(x_t) = \frac{1}{\sqrt{\alpha_t}} \left(x_t - \frac{\beta_t}{\sqrt{1 - \bar{\alpha}_t}} \varepsilon_\theta(x_t, t/T) \right), \tag{29}$$

$$\widetilde{\beta}_t = \frac{1 - \bar{\alpha}_{t-1}}{1 - \bar{\alpha}_t} \beta_t, \tag{30}$$

$$x_{t-1} = \mu_t(x_t) + \sqrt{\widetilde{\beta}_t}\, z_t. \tag{31}$$

The full sampler is a right fold over the finite time indices with an explicit noise stream z_t, so stochasticity is not implicit in the recursive definition.

DDIM and deterministic sampler semantics. The deterministic $\eta = 0$ DDIM step reuses the same $\widehat{x}_0$ reconstruction and recomposes the previous sample by

$$x_{t-1} = \sqrt{\bar{\alpha}_{t-1}}\, \widehat{x}_0 + \sqrt{1 - \bar{\alpha}_{t-1}}\, \varepsilon_\theta(x_t, t/T). \tag{32}$$

TORCHLEAN also exposes DDIM as a real-valued `DynamicalSystem` step. This is useful because the same trajectory, fixed-point, Lipschitz, and contraction vocabulary used for state-space models can be applied to sampler transitions.

Probability-flow ODE. For continuous time, a linear VP schedule stores β_0, β_1 and defines

$$\beta(t) = \beta_0 + t(\beta_1 - \beta_0), \tag{33}$$

$$\bar{\alpha}(t) = \exp\left(-\beta_0 t - \frac{1}{2}(\beta_1 - \beta_0)t^2\right), \tag{34}$$

$$\sigma(t) = \sqrt{1 - \bar{\alpha}(t)}. \tag{35}$$

Under an epsilon parameterization, the probability-flow ODE right-hand side is specified as

$$f_\theta(x, t) = -\frac{1}{2}\beta(t)x + \frac{\beta(t)}{\sigma(t)}\varepsilon_\theta(x, t). \tag{36}$$

The Euler step is the simple total tensor operation

$$E_{\Delta t}(x, t) = x + \Delta t\, f_\theta(x, t), \tag{37}$$

and the sampler integrates backward on a uniform grid with negative Δt. A fixed-time Euler step is exposed as a `DynamicalSystem`, giving samplers a common interface with other discrete dynamical systems in TORCHLEAN.

Sampler boundary and stability theorems. The sampler proof layer records small facts that are important for checkers and model transformations. Zero-step samplers return the initial sample. DDIM and PF-ODE system adapters reduce definitionally to the corresponding step functions. For Euler integration, a generic metric bound has the form

$$\|E_{\Delta t}(x) - E_{\Delta t}(y)\|_2 \le \|x - y\|_2 + |\Delta t|\, \|f(x) - f(y)\|_2. \tag{38}$$

If the right-hand side is L-Lipschitz, then one Euler step is $(1 + |\Delta t|L)$-Lipschitz. The same adapter pattern transports Lipschitz or contractive-step facts to DDIM/PF-ODE systems. These theorems are deliberately modest: they do not prove convergence of a trained denoiser, but they make sampler transformations and verification targets mathematically explicit.

C.5. Probability-theory layer for Gaussian forward processes

The pure diffusion specification treats noise as an explicit input. The probability theory layer adds the mathematical law when the noise is Gaussian. In a finite-dimensional Euclidean space E, let $Z \sim \mathcal{N}(0, I)$ and define

$$X_t = c_0 x_0 + c_1 Z, \qquad c_0 = \sqrt{\bar{\alpha}_t}, \quad c_1 = \sqrt{1 - \bar{\alpha}_t}. \tag{39}$$

The formal object is the push-forward of the standard Gaussian measure under the affine map $z \mapsto c_0 x_0 + c_1 z$. TORCHLEAN proves that this forward law is a probability measure and records the corresponding Gaussian structure for the affine image. Keeping this layer separate is important: tensor samplers can remain total and executable, while probabilistic claims are made only where the relevant measure-theoretic assumptions are available.

This is also the template for future probabilistic generative models. A VAE, normalizing-flow, or stochastic-diffusion theorem should distinguish (i) the deterministic tensor map used by the runtime, (ii) the distribution over the random source, and (iii) the theorem connecting the push-forward or transition kernel to the intended probabilistic model.

C.6. Self-supervised objective algebra and anti-collapse guards

Self-supervised learning is formalized at the objective contract level, covering MAE, JEPA, VICReg, and Barlow-style objective families (He et al., 2022; Assran et al., 2023; Bardes et al., 2022; Zbontar et al., 2021). We do not try to prove that self-supervised training discovers useful representations. Here we record the finite semantics of masks, views, target branches, predictive losses, and geometry guards, so an experiment or theorem cannot silently change the objective being optimized.

Finite masks and masked loss. For n patches or tokens, a mask is a predicate $m : \mathrm{Fin}(n) \to \mathrm{Bool}$. The selected-index form is a list $I = [i_1, \ldots, i_k]$, and the generic masked loss is

$$\mathrm{maskedLoss}(I, \ell) = \sum_{i \in I} \ell(i). \tag{40}$$

Lean proves the elementary list laws needed to treat the serialized mask as a finite objective: nil is zero, cons adds one term, appending index lists adds losses, reversing the index list preserves the sum, and if every selected per-index loss is zero then the masked loss is zero.

Listing 13. Schematic masked objective used by MAE and JEPA-style losses.

```
def maskedLoss (idxs : List (Fin n)) (
    lossAt : Fin n -> Float) : Float :=
  idxs.foldr (fun i acc => lossAt i + acc)
    0.0

-- The concrete proof is by induction on xs
    .
theorem maskedLoss_append (xs ys : List (
    Fin n)) (lossAt : Fin n -> Float) :
    maskedLoss (xs ++ ys) lossAt =
    maskedLoss xs lossAt + maskedLoss ys
    lossAt
```

MAE. For a finite patch batch $x : \mathrm{Fin}(n) \to \mathrm{Patch}$, predictions p_i, and per-patch loss ℓ, the MAE objective is

$$\mathrm{MAE}(I, x, p) = \sum_{i \in I} \ell(x_i, p_i). \tag{41}$$

The formal theorems state that this loss decomposes over appended mask lists, is invariant under reversal of I, is zero when every selected patch loss is zero, and that identity decoding gives exact reconstruction. These are small facts, but they pin down the intended semantics: masked reconstruction is a set-like objective over selected patches, not an artifact of an arbitrary serialization order.

JEPA and target-branch extensionality. JEPA-style objectives separate a context representation c, target representations t_i, a predictor $p(c, i)$, and a representation loss d. The finite objective is

$$\mathrm{JEPA}(I, c, t, p) = \sum_{i \in I} d(t_i, p(c, i)). \tag{42}$$

The target branch is modeled as an ordinary value at the objective boundary, matching the stop-gradient design intent. The theorem `jepaLoss_target_ext` states that if two target branches agree on selected indices, then the loss is identical. Thus unselected target coordinates cannot influence the objective.

Predictive-view contract. The generic contract unifies MAE and JEPA. A `PredictiveViewContract` stores target indices I, a context value, a target value at every finite index, a target encoder, a predictor, a per-index distance, and a nonnegative geometry guard. The objective is

$$\mathrm{Obj}(C) = \mathrm{PredictiveLoss}(C) + \mathrm{GeometryGuard}(C). \tag{43}$$

MAE is recovered by choosing the identity target encoder into patch/pixel space and zero geometry guard. JEPA is recovered by choosing the latent target representation as the target space. Lean proves both instance theorems: the predictive-view loss/objective reduces definitionally to the MAE or JEPA finite objective under these choices.

VICReg, Barlow-style guards, and view graphs. Geometry guards are added orthogonally to the prediction term. VICReg-style summaries include invariance, variance-floor, and covariance/redundancy components with weights (λ, μ, ν). Barlow-style summaries penalize off-diagonal redundancy and diagonal mismatch. The current finite theory records local anti-collapse facts: collapsed coordinates pay a positive variance-floor penalty under a positive margin, the identity correlation summary has zero redundancy loss, and collapsed diagonal summaries incur a positive Barlow-style penalty. A view-graph energy layer represents finite alignment objectives

$$E_G(z) = \sum_{(i,j) \in E(G)} \|z_i - z_j\|^2, \tag{44}$$

which is zero for collapsed representations but becomes useful only when paired with a spread or variance guard. This makes the common SSL distinction precise: alignment alone permits collapse; the geometry guard is the formal object that rules it out locally.

C.7. Sequence, generative, scientific, and diagnostic model families

The examples now cover more than the verifier core. The sequence layer includes causal attention, multi-head attention, RoPE-style positional rotations, token/embedding boundaries, generation loops, recurrent layers, and selective-scan primitives for Mamba/S4-style state-space models. The central semantic issue is causality: future tokens should not change prefix outputs, cache offsets must agree with token positions, and chunked or fused scans should refine the same recurrence. TORCHLEAN exposes these as formal contracts and as regression tests. For state-space models, scan composition lemmas connect optimized block scans to sequential recurrence, and prefix/non-anticipation lemmas state that appending future inputs preserves emitted prefix outputs.

The generative and scientific layers use the same tensor language for different mathematical roles. Diffusion examples expose schedules, denoisers, sampler states, and reverse transitions. VAE/VQ-VAE/GAN-style examples expose latent variables, diagonal-Gaussian or KL terms, finite codebooks, nearest-code quantization, reconstruction losses, and generator/discriminator objectives. FNO/PINN examples use spectral operators, derivatives, and residual constraints; spline/ODE and Arb-backed workflows use external numerical producers followed by Lean-side certificate replay. We include these examples to show that modern ML workflows can be assigned named semantic objects that are suitable for later checking; the section is not a claim about leaderboard model quality.

The diagnostic layer makes semantic boundary failures explicit. The Bug Zoo covers cases such as causal-mask leakage, KV-cache/RoPE position drift, stable-loss boundaries, tokenizer/embedding mismatches, batch-invariance bugs, normalization-state bugs, Float32 edge cases, padding conventions in native kernels, and certificate/IR import mismatches. Each bug-shaped test is small on purpose: it isolates a boundary where a conventional ML system can silently change semantics. In TORCHLEAN, the corresponding property can be stated as a mask theorem, prefix-invariance check, schema check, shape invariant, finite-precision contract, or certificate replay obligation.

D. Numerical Semantics

Motivation and trust boundaries. We make numerical semantics explicit because it is easy to prove theorems about a real-valued model and then silently execute a

Float32 implementation whose behavior differs at exactly the corner cases that matter for verification (rounding, overflow/underflow, NaN/Inf propagation, signed zeros, and library conventions). IEEE 754 is the de facto standard for floating-point arithmetic: it specifies binary/decimal formats, rounding rules, and exception behavior (including NaNs/Infs and their default handling). In Lean, however, the built-in runtime floating-point types are *opaque to the kernel*: they are intended for computation and are implemented by external runtime code rather than reducible definitions in the logic (Lean Prover Community, 2025). As a result, we treat the scalar type α as an explicit parameter and require every theorem or executable demo to declare which numeric semantics it is using; this turns "what arithmetic are we reasoning about?" into part of the statement rather than an implicit convention.

IEEE 754 (what it standardizes). IEEE 754 defines floating-point numbers as signed significands with bounded exponents (e.g., binary32/Float32), together with *rounding modes* (typically round-to-nearest-even) and *exceptional values* such as $\pm\infty$ and NaNs. It also specifies default behaviors for exceptional operations (e.g., division by zero, invalid operations) and comparison/order conventions. For verification, many "real-analysis proofs" do not apply verbatim in IEEE arithmetic: operations are rounded, may overflow/underflow, and may produce non-finite values whose propagation rules are part of the semantics.

Motivation from Flocq. Our design is inspired by Flocq, a mature Roq library for reasoning about floating-point arithmetic. Flocq cleanly separates (i) the *format* (which numbers are representable: radix, exponent bounds, subnormals) from (ii) the *rounding operator* (how an exact real result is mapped to a representable value), and it provides theorem infrastructure for compositional error bounds. (Boldo & Melquiond, 2011) We adopt this same separation because neural network verification needs both: *(a)* theorem-friendly "round-on-$\mathbb{R}$" models to state and compose numerical error envelopes, and *(b)* executable, bit-level semantics to make corner cases (NaNs, signed zeros, overflow) concrete when running full demos and checkers.

Summary: four numeric layers in TORCHLEAN. We provide (i) *trusted runtime floats* for speed, (ii) an *executable IEEE-754 model* for explicit Float32 behavior, (iii) *proof level rounding models* for compositional error reasoning, and (iv) an *external validated-numerics oracle* for rigorous high-precision enclosures of difficult primitives (notably transcendentals).

(A) Runtime execution: `Float`/`Float32` (fast, but opaque). Lean exposes `Float` (binary64) and `Float32` (binary32) as runtime types intended for efficient execution. Their operations are implemented by external/runtime code (e.g., compiled to underlying machine/C operators) rather than by kernel-reducible definitions, so we treat them as an *explicit trust boundary* for proofs. (Lean Prover Community, 2025)

(B) Executable Float32 semantics: `IEEE32Exec` (bit-level IEEE). For full demos that require explicit Float32 behavior, we implement `IEEE32Exec`, a Lean-defined, bit-level IEEE-754 binary32 kernel. This model includes signed zeros, subnormals, NaNs/Infs, and the IEEE rounding/exception behavior for core arithmetic. It is *executable* inside Lean and therefore gives an explicit meaning to "Float32 execution" independent of a particular runtime implementation. The remaining "hardware gap" (showing a target platform refines this exact software model) is handled as an explicit conformance interface rather than being silently assumed.

(C) Proof-relevant rounding models: `FP32`/`NF` (round after each primitive). For theorem statements and error analysis, we use "round-on-$\mathbb{R}$" models inspired by Flocq: each primitive is specified as "compute in $\mathbb{R}$, then round" under a chosen format/rounding mode, enabling compositional error envelopes through entire graphs. `FP32` instantiates this with IEEE-style binary32 rounding (finite-only, i.e., it does not model NaN/Inf), while `NF` generalizes the same idea to user-chosen radices/exponent functions/rounding operators.

(D) Validated numerics oracle: Arb/FLINT (rigorous transcendental enclosures). For rigorous enclosures of transcendental functions (e.g., $\tanh, \exp, \log, \sqrt{\cdot}, \sin, \cos$) at user-chosen precision, we integrate an Arb/FLINT backend via `python-flint` behind an explicit oracle boundary. The oracle returns certified ball/interval enclosures at a specified working precision (in bits), which we can consume as certificate-like artifacts in verification pipelines. This layer is *not* kernel-reducible and is treated as an external trust boundary; its role is to provide high-quality validated bounds where implementing tight transcendental enclosures natively is cumbersome, while keeping the trusted computing base explicit.

Why we implement *both* (bit-level and round-on-ℝ). These layers serve different proof/verification needs: *bit-level execution* (`IEEE32Exec`) is ideal for making "what happens on Float32?" concrete in checkers and demos, including edge cases (NaNs, signed zeros) that real analysis ignores; *round-on-ℝ models* (`FP32`/`NF`) are ideal for theorem statements and compositional error envelopes because they expose the rounding operator as a mathematical object that can be bounded and composed (in the style of Flocq). (Boldo & Melquiond, 2011) Together, they let us support both the verification community (explicit executable semantics) and the floating-point proof community (compositional rounding/error reasoning) under a single semantic umbrella.

Lean-specific note (why opacity matters). Because Lean's runtime floats are not encoded in the logic, the kernel cannot reduce or reason about them without additional axioms; in particular, floating-point operations are implemented externally (and thus are not definitionally equal to any mathematical model inside Lean). This is why TORCHLEAN separates "fast execution" from "proved semantics" and makes the trust boundary explicit in both Section 1 and Table 1.

One surface name, switchable semantics. To keep model code uniform while making numerical assumptions explicit, TORCHLEAN exposes a single surface notion of "Float32" with a selectable semantic mode. This lets the same model/program be instantiated with (i) an executable bit-level IEEE-754 semantics for full runs, or (ii) a proof level rounding model for theorem statements and compositional error envelopes, without rewriting the model.

Listing 14. Selecting Float32 semantics at the type level.

```
open TorchLean.Floats

-- Executable Float32 semantics (bit-level
    IEEE-754 binary32):
abbrev ExecF32 : Type := F32
      -- = IEEE32Exec

-- Proof Float32 semantics (round-on-R,
    finite-only):
abbrev ProofF32 : Type := F32 .fp32
      -- = FP32
```

Why multiple float modes are necessary. No single floating-point representation serves all verification goals. IEEE 754 defines the concrete behavior of deployed floating-point arithmetic, including rounding and exceptional values (NaNs/Infs, signed zeros, subnormals). For *executable demos* and certificate checking where these corner cases matter, we want an explicit, runnable model of binary32 semantics—hence `IEEE32Exec`. For *theorem statements* about numerical stability and graph level error budgets, we instead want a proof-level "round after each primitive" model over ℝ that supports compositional error reasoning in the style of verified floating-point libraries such as Flocq—hence `FP32`/`NF`. Finally, Lean's built-in runtime floats (`Float`/`Float32`) are fast but opaque to the kernel and therefore live on the explicit trust/validation side of the interface.

Float32 mode selection. At the API level we package this choice behind a small mode enum, so use sites are explicit about which semantics they rely on:

Listing 15. A single "Float32" name with explicit semantics.

```
inductive Float32Mode where
  | fp32 | ieee754Exec

abbrev Float32 (mode : Float32Mode := .
    ieee754Exec) : Type :=
  match mode with
  | .fp32        => FP32
  | .ieee754Exec => IEEE32Exec
```

Proof-level rounding model (`NF`). For graph level error bounds we use a proof level model that rounds after every primitive operation. Conceptually, an `NF` value stores a real number together with a chosen *format* (radix and exponent function) and *rounding* operator; each primitive is specified as "compute in ℝ, then round," and local rounding/error lemmas compose over SSA/DAG graphs.

Practical rule of thumb. Use `IEEE32Exec` when you want executable Float32 behavior (including corner cases) to match an IEEE-style semantics; use `FP32`/`NF` when you want theorem statements with explicit, compositional rounding error envelopes; use runtime `Float`/`Float32` for fast prototyping where the numeric backend is an explicitly trusted assumption.

Listing 16. Schematic runtime conformance contract for deployed Float32.

```
-- A theorem using this class is
    conditional on the deployment target.
class RuntimeFloat32MatchesIEEE32Exec :
    Prop where
  add_finite : forall x y,
    finite x -> finite y -> finite (x + y)
    ->
    toIEEE32 (runtimeAdd x y) = IEEE32Exec.
    add (toIEEE32 x) (toIEEE32 y)
  mul_finite : forall x y,
    finite x -> finite y -> finite (x * y)
    ->
    toIEEE32 (runtimeMul x y) = IEEE32Exec.
    mul (toIEEE32 x) (toIEEE32 y)
  relu_exact : forall x,
    toIEEE32 (runtimeRelu x) = IEEE32Exec.
    max (toIEEE32 x) IEEE32Exec.posZero
```

We use this form intentionally. A graph theorem can be unconditional about `IEEE32Exec`, while a deployed-

runtime theorem carries a visible assumption such as `RuntimeFloat32MatchesIEEE32Exec`. The assumption records the target configuration: rounding mode, denormal policy, fused operations, reduction order, and library implementation choices.

D.1. Whole-graph NF bounds and hardware soundness

This section explains two complementary pieces of our numerical story: (i) how we *compose* per-operator rounding/error lemmas into graph level error budgets in a "round after each primitive" model (NF), and (ii) how we connect an *executable* bit-level Float32 semantics to that proof model on the finite (non-NaN/non-Inf) path, and what remains to relate either of them to *hardware* execution.

Whole-graph NF bounds: local errors compose over SSA/DAG. NF is a proof level "round after each primitive" semantics: each primitive is specified as "compute in $\mathbb{R}$, then round" under a chosen format/rounding operator. We relate NF values to real semantics via an explicit error relation $\approx_\varepsilon$, and we prove graph level bounds by induction in SSA/topological order (and analogously for backward sweeps when needed).

Local-to-global error propagation (NF idea). Define a scalar approximation relation

$$x_{\mathrm{nf}} \approx_\varepsilon x_{\mathbb{R}} \quad \Longleftrightarrow \quad |x_{\mathrm{nf}} - x_{\mathbb{R}}| \le \varepsilon,$$

and lift it to tensors (pointwise or via a norm such as ℓ_∞). Each primitive operator op comes with a lemma of the form:

$$\text{if } u_j \approx_{\varepsilon_j} v_j \text{ for all inputs } j, \text{ then } \mathrm{op}_{\mathrm{nf}}(u) \approx_{\varepsilon'} \mathrm{op}_{\mathbb{R}}(v),$$

where the derived ε' is computed from operator-specific stability bounds. For linear/affine primitives, ε' follows from standard Lipschitz constants; for nonlinear primitives, it follows from local derivative bounds or global Lipschitz envelopes (on the region of interest). Whole-graph bounds are proved by induction over the SSA/DAG order: the induction hypothesis provides bounds for parent nodes, and the node lemma yields the bound for the current node.

Executable IEEE model $\to$ rounding-on-$\mathbb{R}$ model (internal refinement). NF/FP32-style reasoning is theorem-friendly because rounding is an explicit mathematical operator on $\mathbb{R}$, but it does not capture IEEE corner-case behavior (NaNs/Infs, signed zeros, subnormals) directly. Conversely, a bit-level IEEE model is executable and makes those corner cases concrete, but it is harder to use for compositional error proofs. We therefore establish an *internal refinement* on the *finite* path: for the core Float32 arithmetic primitives (addition/subtraction/multiplication/-division/sqrt and $\min / \max$), we prove that interpreting the executable bit-level result as a real number agrees with applying the corresponding round-to-Float32 operator to the real arithmetic result, under explicit side conditions that rule out NaN/Inf and overflow-to-Inf.

Theorem D.1: Internal refinement on the finite path

Let ExecOp_{32} be a core Float32 primitive implemented by the executable bit-level IEEE model (e.g., $+, -, \times, /, \sqrt{\cdot}$, $\min / \max$), and let RealOp be the corresponding real operator. (For $\min / \max$ on finite values, $\mathrm{ExecOp}_{32}(x, y)$ returns exactly one of its inputs, so the Round_{32} wrapper below is vacuous; we state the refinement uniformly for all listed primitives.) Let toReal interpret a finite Float32 value as a real number, and let Round_{32} be the IEEE-style round-to-Float32 operator on $\mathbb{R}$. Define $r(x, y) := \mathrm{RealOp}\big(\mathrm{toReal}(x), \mathrm{toReal}(y)\big)$. Assume the evaluation stays *finite* (no NaN/Inf and no overflow-to-Inf) on the considered inputs. Then for all inputs x, y in the finite domain,

$$\mathrm{toReal}\big(\mathrm{ExecOp}_{32}(x, y)\big) = \mathrm{Round}_{32}\big(r(x, y)\big).$$

Moreover, these per-primitive refinements compose: for any expression e built from the core primitives, interpreting the executable result with toReal agrees with evaluating the corresponding real expression and rounding at each primitive, under the same finiteness assumptions.

This bridge is necessary because it lets us run full demos under an explicit IEEE-754 semantics while still reusing the cleaner FP32/NF error-envelope lemmas whenever the execution stays in the finite regime.

Hardware soundness: what remains and a pragmatic deployment path. IEEE 754 specifies formats, rounding modes, and exception behavior, but real deployments can diverge through compilation and platform choices (e.g., flush-to-zero/denormals-are-zero, reassociation/"fast-math", FMA contraction, or reduction order). To claim that *hardware* `Float32` execution inherits our theorems, one must additionally (a) fix a target semantics contract (rounding mode, denormal policy such as FTZ/DAZ, contraction/reassociation policy, and reduction ordering), and (b) relate the compiled runtime's observable Float32 results to the chosen executable model (or to the rounding-on-$\mathbb{R}$ model) for the operations actually used. This remains target-level future work.

Transcendentals and IEEE-754 (what is and is not specified). IEEE 754 precisely specifies formats and

core arithmetic (e.g., add/sub/mul/div, sqrt, comparisons, NaN/Inf behavior), but *elementary/transcendental functions* (e.g., `exp`, `log`, `sin`, `cos`, `tanh`) are largely outside the standard's required semantics and are typically provided by system `libm` implementations. [1] As a result, even when two platforms are "IEEE compliant" for core arithmetic, their transcendental results can differ across OS/compiler/libm versions or under different optimization flags. A deeper reason is the *table-maker's dilemma*: deciding the correctly-rounded result for an elementary function can require substantially more precision than the target format in worst cases, which makes full bit-exact specification and implementation nontrivial in practice. (Brisebarre et al., 2025; Zimmermann, 2005)

Our deliberate split: executable IEEE core vs. explicit transcendental policy. Accordingly, TORCHLEAN separates concerns. For core IEEE arithmetic, `IEEE32Exec` provides a Lean-defined, bit-level model of binary32 behavior (including signed zeros, subnormals, NaN/Inf propagation, and rounding), so the meaning of "Float32 execution" is explicit inside the prover. For transcendental functions, we make the policy explicit rather than pretending it is uniquely determined by IEEE:

- *Deterministic in-kernel implementations for common ML primitives.* For functions heavily used in ML pipelines (notably `exp`/`log` and hyperbolic functions used in activations and normalizers), we provide deterministic implementations with a fixed rounding/approximation policy so that full executions are reproducible under `IEEE32Exec`.
- *Explicit delegation when necessary.* For functions outside the verified kernel surface (e.g., full trigonometric stacks), we may delegate to Lean runtime `Float` (or a chosen library implementation) and then round back to binary32, treating that choice as an *explicit trust boundary* rather than an implicit semantic fact.

Proof models cover transcendentals via "round the real function." In parallel, the FP32/NF proof models include transcendentals definitionally as "apply the real function, then round," which supports theorem statements in terms of explicit error envelopes and interval enclosures under stated hypotheses. This mirrors the classical verified-float approach: instead of depending on a particular `libm` implementation, the semantics is a mathematical rounding operator applied to the real function, enabling compositional reasoning. (Brisebarre et al., 2025; Flocq Developers, 2025)

NaN payload caveat (why hardware conformance is subtle). IEEE 754 specifies NaN propagation at a high level (operations produce NaN when given NaN inputs), but the *choice of NaN payload* (and some signaling-vs-quiet details) is not fully uniform across real implementations. (IEEE Standards Association, 2019) Our executable kernel fixes a deterministic policy (including a specific quieting/selection rule), and we prove properties about that policy. Consequently, any claim that a concrete compiler/runtime/hardware Float32 implementation refines `IEEE32Exec` must assume or establish compatibility with this concrete NaN policy (or adopt a quotienting notion of observational equivalence that treats NaN payload differences as irrelevant for the target theorems).

[1] IEEE's own background material explicitly notes that many programs rely on library elementary functions and that the standard does not specify them.

E. CROWN verification: bounds, duality, and certificates

We expand Section 2 with implementation details for bound propagation and certificate checking.

E.1. IBP, CROWN/LiRPA, and certificate checking

Overview. Our verifier operates on the shared operator-tagged SSA/DAG IR and proves properties of the graph denotation $[\![G]\!]$ by establishing *sound enclosures* for intermediate values and/or outputs. We support two complementary verification modes: *(i) native bound propagation* implemented in Lean for demo-scale graphs, and *(ii) certificate checking*, where an external verifier produces a bound/certificate that Lean checks against the same IR semantics.

Acronyms.

- **IBP** (Interval Bound Propagation): propagate interval boxes $[l, u]$ forward through the network.
- **CROWN**: propagate *linear* upper/lower bounds through nonlinearities to obtain tighter output bounds.
- **LiRPA** (Linear Relaxation-based Perturbation Analysis): a general framework that derives/propagates linear relaxations (including CROWN- and DeepPoly-style rules) over general computational graphs.
- **α-CROWN**: tightens CROWN bounds by optimizing relaxation parameters.
- **β-CROWN**: incorporates split constraints (branch-and-bound) efficiently by encoding splits with optimizable parameters.

IBP (interval bound propagation). Interval Bound Propagation (IBP) is the simplest sound enclosure method for neural computation graphs. Starting from an *input*

region—typically an axis-aligned box $[l_0, u_0]$ (often used to over-approximate an ℓ_∞ ball)—IBP computes, for every node i in the graph, an interval enclosure $[l_i, u_i]$ such that the true node value satisfies $v_i(x) \in [l_i, u_i]$ for all admissible inputs $x \in [l_0, u_0]$. The algorithm is a single forward sweep in topological order: each primitive provides an *interval transfer rule* that maps parent bounds to a sound output bound, maintaining the invariant "parents sound $\Rightarrow$ child sound." IBP is attractive because it is fast, compositional, and easy to make formally sound, but it can be loose because plain intervals do not track correlations between coordinates (and thus can over-approximate significantly after repeated mixing through linear layers).

Example IBP transfer rules (sketch). All rules below are *sound*: they guarantee $\mathrm{op}(v) \in [l', u']$ for all $v \in [l, u]$.

- **Affine / linear** $y = Wx + b$ (includes `linear` and flattened `matmul` cases). Write $W = W^+ - W^-$ with $W^+, W^- \geq 0$ elementwise. Then

$$l_y = W^+ l_x - W^- u_x + b, \qquad u_y = W^+ u_x - W^- l_x + b,$$

  which is the standard interval-arithmetic enclosure for linear maps.

- **Monotone elementwise** (e.g., ReLU, exp). Apply the function to endpoints:

$$[l_y, u_y] = [f(l_x), f(u_x)] \quad \text{(componentwise)}.$$

  For ReLU specifically, this yields $l_y = \max(l_x, 0)$ and $u_y = \max(u_x, 0)$.

- **Reductions** (e.g., sum/mean over a fixed axis). Because reduction is linear, bounds reduce by summing/averaging endpoints along the axis:

$$\mathrm{sum}([l, u]) = [\mathrm{sum}(l), \mathrm{sum}(u)]$$

$$\mathrm{mean}([l, u]) = [\mathrm{mean}(l), \mathrm{mean}(u)].$$

Listing 17. Schematic interval objects and the linear IBP step.

```
structure Box (s : Shape) where
  lo : Tensor Float s
  hi : Tensor Float s

-- W = Wpos - Wneg, with Wpos,Wneg >= 0
    elementwise.
def ibpLinear (W b : Tensor Float _) (x :
    Box inShape) : Box outShape :=
  let Wpos := Tensor.map (fun w => max w
    0.0) W
  let Wneg := Tensor.map (fun w => max (-w)
     0.0) W
  { lo := Wpos.matmul x.lo - Wneg.matmul x.
    hi + b,
    hi := Wpos.matmul x.hi - Wneg.matmul x.
    lo + b }

def ibpRelu (x : Box s) : Box s :=
  { lo := Tensor.map (fun z => max z 0.0) x.
    lo,
    hi := Tensor.map (fun z => max z 0.0) x.
    hi }
```

The Lean checker does not need to trust the producer's explanation of how a bound was found. It recomputes the local transfer step from parent boxes and compares the recomputed box with the claimed box. The following schematic pattern is the core of the checker.

Listing 18. Schematic replay step for a node certificate.

```
def checkNode? (G : Graph) (cert : Cert) (
    st : CertState) (i : Nat) :
    Except CheckError CertState := do
  let node      <- G.node? i
  let parents   <- st.lookupAll node.
    parents
  let expected  <- transfer node.kind
    parents
  let claimed   <- cert.boxFor i
  guard (sameShape expected claimed)
  guard (claimed.lo <= expected.lo + cert.
    tol)
  guard (expected.hi <= claimed.hi + cert.
    tol)
  pure (st.insert i claimed)
```

The inequalities are intentionally one-sided. A certificate may be looser than the recomputed bound and still be sound; it may not claim a lower bound above the recomputed lower bound or an upper bound below the recomputed upper bound unless the checker can prove that stronger claim.

IBP certificate soundness (graph dialect). To support certificate checking, we formulate IBP soundness for a *safe* graph semantics that is explicitly partial: node evaluation fails when required parent values/parameters are missing or when declared dimensions do not match. We then define a per-node IBP *certificate step* that deterministically recomputes each node's interval box from its parents' boxes. The soundness proof follows the standard local-to-global pattern: under a topological order and a supported operator subset, local semantic consistency and local certificate consistency imply a global enclosure invariant.

Theorem E.1: IBP certificate soundness

Let G be a topologically sorted computation graph over a supported operator fragment, and fix a parameter store. Assume we are given: *(i)* a partial value semantics that computes node values by a one-step rule from parent values and parameters, and *(ii)* a one-step

IBP rule that computes an interval box for each node from its parents' boxes and parameters. Let $(v_i)_i$ be any assignment of semantic node values that is locally consistent with the one-step value rule, and let $(B_i)_i$ be any assignment of certified interval boxes that is locally consistent with the one-step IBP rule. If each semantic input lies in its declared input box, then for every node i, whenever both v_i and B_i are defined, the value v_i lies in B_i componentwise.

Whole-graph IBP soundness for the concrete implementation. Beyond certificate soundness, we also connect the theorem to the concrete implementations used in the demos. We define total "evaluate-by-id" and "propagate-by-id" procedures, implemented by recursion on node id, and prove that they satisfy the local-consistency premises of Theorem E.1. As a result, the computed IBP boxes enclose the computed semantic values whenever both procedures return values:

Theorem E.2: Whole-graph IBP soundness for `runIBP?`

Let `evalGraphRec` be the total value evaluator for G and `runIBP?` the corresponding total IBP propagation. Under the same structural assumptions (topological order, supported operators, and input enclosure), for every node i, whenever both procedures return a value v_i and a box B_i, we have $v_i \in B_i$ componentwise.

Proof idea (both theorems). The enclosure invariant is preserved one node at a time: assuming all parent enclosures hold, the operator-specific IBP transfer rule yields an enclosure for the current node consistent with its value semantics. Topological order ensures that, at node i, all parent facts are already established, so the global result follows by induction over node IDs/topological order.

CROWN and LiRPA (linear relaxations). CROWN tightens IBP by tracking *affine bounds* rather than pure intervals: instead of only maintaining $v_i(x) \in [l_i, u_i]$ for each node, it maintains linear forms that bound each node as a function of the input, e.g. $a^\top x + b \le v_i(x) \le c^\top x + d$ over the admissible input region. These affine bounds are obtained by replacing nonlinear primitives with *sound linear envelopes* on the current pre-activation interval and propagating the resulting affine forms through linear operators. For piecewise-linear activations such as ReLU, the envelope is given by valid upper/lower lines on $[l, u]$ (secant/tangent choices in the unstable regime $l < 0 < u$); for smooth activations (e.g., $\tanh$, sigmoid), one uses tangent/secant bounds or other valid global/region-restricted linear relaxations, yielding a strictly tighter enclosure whenever correlations between coordinates matter.

LiRPA is the unifying viewpoint: it treats bound propagation as computing and composing such linear relaxations over *general computational graphs* (not just simple feedforward chains), subsuming CROWN- and DeepPoly-style rules. Modern LiRPA implementations typically expose both (i) a *forward* pass that produces node wise affine bounds and (ii) an *objective-dependent* backward/dual pass that tightens the bound for a specific linear objective on the output (e.g., a robustness margin objective), since the best relaxation choices can depend on the downstream objective. In TORCHLEAN, our Lean-based core mirrors this structure: we provide a proved-sound IBP layer and a basic CROWN/LiRPA affine engine over the shared operator-tagged IR, with conservative fallbacks (e.g., deriving constant affine bounds from IBP) when a specialized relaxation is not yet implemented, preserving soundness at the cost of tightness.

Example: ReLU linear relaxation (one neuron). Let $z \in [l, u]$ and $y = \mathrm{ReLU}(z)$. A standard sound relaxation chooses linear bounds

$$y \ge \lambda_L z + \mu_L, \qquad y \le \lambda_U z + \mu_U,$$

with coefficients chosen so the lines bound ReLU on $[l, u]$ (e.g., $\lambda_U = \frac{u}{u-l}$, $\mu_U = -\frac{ul}{u-l}$ for the secant upper bound when $l < 0 < u$; and a valid lower bound selected from the admissible family in this regime). Propagating these envelopes yields affine bounds on downstream nodes and, ultimately, output enclosures.

Listing 19. Schematic ReLU relaxation coefficients used by CROWN.

```
structure ReluRelax where
  lowerSlope     : Array Float
  lowerIntercept : Array Float
  upperSlope     : Array Float
  upperIntercept : Array Float

def reluRelax (lo hi : Tensor Float s) :
    ReluRelax :=
  Id.run do
    let mut ls := #[]; let mut li := #[]
    let mut us := #[]; let mut ui := #[]
    for k in [:lo.numel] do
      let l := lo.data[k]!; let u := hi.
    data[k]!
      if 0.0 <= l then
        ls := ls.push 1.0; li := li.push
    0.0
        us := us.push 1.0; ui := ui.push
    0.0
      else if u <= 0.0 then
        ls := ls.push 0.0; li := li.push
    0.0
        us := us.push 0.0; ui := ui.push
    0.0
      else
```

```
        let secant := u / (u - l)
        let offset := -(l * u) / (u - l)
        -- Upper line is the secant. Lower
    line is chosen from
        -- a sound family, possibly
    optimized by alpha-CROWN.
        us := us.push secant; ui := ui.push
     offset
        ls := ls.push 0.0;    li := li.push
     0.0
    return { lowerSlope := ls,
    lowerIntercept := li,
             upperSlope := us,
    upperIntercept := ui }
```

α/β-CROWN as certificate-checked bound semantics (graph dialect). We implement an α/β-CROWN certificate interface for the operator-tagged graph verifier dialect. Fix a compiled graph G (nodes i in topological order) and an input box B. CROWN-family bounds at each node are represented as *affine enclosures* over inputs: for a scalar node value z_i we enclose it by two affine functions of the input x,

$$z_i(x) \in [L_i(x), U_i(x)],$$
$$L_i(x) = \langle a_i^L, x\rangle + b_i^L,$$
$$U_i(x) = \langle a_i^U, x\rangle + b_i^U.$$

(For tensor nodes, the same form is applied componentwise.) Bounds are computed by a per-node step rule

$$\texttt{CrownStepNode?} : S \times i \to \mathsf{Option}(\mathsf{AffBounds}),$$

where S denotes the checker state (context plus parent bounds) and AffBounds abbreviates the pair of affine maps above. This step extends α-CROWN by optionally incorporating β phase information at ReLU nodes: phase-fixed units use exact linear behavior (slope 0 or 1), while unstable units fall back to standard CROWN/α-CROWN relaxations.

Certificate contents and the role of α and β. An α/β-CROWN certificate provides, for each node i: (i) an IBP pre-activation interval $[l_i, u_i]$ (a local box enclosure); (ii) affine bounds (lower and upper affine maps of the input); (iii) α parameters for the unstable-ReLU *lower* relaxation; and (iv) an optional β phase vector for ReLU units. Intuitively, α selects a member of a sound lower-envelope family in the unstable regime ($l < 0 < u$), while β encodes phase constraints (active/inactive) that, when consistent with IBP, permit *exact* linear behavior on those units.

β phases and phase consistency. For a ReLU node with pre-activation z and post-activation $y = \mathrm{ReLU}(z)$, we interpret $\beta \in \{-1, 0, 1\}$ as

$$\beta \equiv \begin{cases} -1 & \text{inactive } (z \le 0), \\ 0 & \text{unstable (no constraint)}, \\ 1 & \text{active } (0 \le z). \end{cases}$$

The step checks *phase consistency* against the IBP pre-activation interval $[l, u]$:

$$\mathsf{Consistent}(l, u, \beta) \;:\Longleftrightarrow\; (\beta = -1 \Rightarrow u \le 0) \;\wedge\; (\beta = 1 \Rightarrow 0 \le l).$$

If consistent, inactive/active phases use exact linearization (slope 0 or 1). Unstable units ($\beta = 0$) fall back to the usual CROWN upper relaxation and α-CROWN lower relaxation.

Phase-dependent ReLU relaxations. Write a linear bound as a pair (s, b) representing the function $sz + b$. Define phase-dependent relaxations by

$$\overline{r}_\beta(l, u) = \begin{cases} (0, 0) & \text{if } \beta = -1 \text{ and } u \le 0, \\ (1, 0) & \text{if } \beta = 1 \text{ and } 0 \le l, \\ \overline{r}(l, u) & \text{if } \beta = 0, \end{cases}$$

$$\underline{r}_{\alpha,\beta}(l, u, \alpha) = \begin{cases} (0, 0) & \text{if } \beta = -1 \text{ and } u \le 0, \\ (1, 0) & \text{if } \beta = 1 \text{ and } 0 \le l, \\ \underline{r}_\alpha(l, u, \alpha) & \text{if } \beta = 0. \end{cases}$$

Here $\overline{r}(l, u)$ is the standard CROWN triangular *upper* relaxation (secant in the crossing case), and $\underline{r}_\alpha(l, u, \alpha)$ is the α-CROWN *lower* relaxation family (with $\alpha \in [0, 1]$ used only when $l < 0 < u$). In the inactive/active cases the relaxations reduce to the exact affine graphs of ReLU on the corresponding half-line.

Operator-level soundness (ReLU, β-aware). Let $x \in [l, u]$ and assume $\mathsf{Consistent}(l, u, \beta)$. Let $(\overline{s}, \overline{b}) = \overline{r}_\beta(l, u)$ and $(\underline{s}, \underline{b}) = \underline{r}_{\alpha,\beta}(l, u, \alpha)$ (with $0 \le \alpha \le 1$), the scalar relaxations satisfy:

$$\mathrm{ReLU}(x) \;\le\; \overline{s}\, x + \overline{b} \text{ and } \underline{s}\, x + \underline{b} \;\le\; \mathrm{ReLU}(x).$$

Certificate checking: replay-based producer/checker design. Certificate checking is the key mechanism for obtaining tight bounds without enlarging the trusted computing base to include a complex optimizer. An external verifier or solver acts purely as an *untrusted producer*: it searches for a certificate (bounds, affine envelopes, split structure, dual variables) and serializes the result as a compact JSON artifact containing per-node IBP boxes, affine bound coefficients, and optional per-node α and β data.

Lean then acts as the *trusted checker*: it parses the artifact, canonicalizes all numeric data to a fixed float

grid, and *recomputes* each node's bound via the same step semantics that defines $[\![G]\!]$, producing $B_i^{\text{Lean}} :=$ `CrownStepNode?`$(\ldots, i) \in$ Option(AffBounds) for each node i in topological order. The checker accepts only if (i) provided bounds match the recomputed bounds after canonicalization, (ii) parent bounds appear in topological order, and (iii) shapes and operator tags are consistent with the IR. The external optimizer is never trusted: regardless of what search or heuristics it uses internally, only the final artifact crosses the trust boundary, and every claim is independently replayed in Lean. This reduces the trusted computing base to the IR denotation plus the small checker.

What the checker validates at each node. For each node i, the checker performs three checks in sequence. First, it validates the *schema*: node i exists in the graph G, its declared input and output shapes match the IR typing, and all required parent nodes have already been processed. Second, it validates *local soundness*: the certified bound B_i is implied by the parent bounds and the `OpKind`-specific enclosure rule, confirmed by comparing B_i against the recomputed B_i^{Lean} after float-grid canonicalization. Third, after all nodes are processed, it validates *goal reduction*: the target property (e.g., a robustness margin, a Lyapunov decrease condition, or a PINN residual bound) follows from the certified output enclosure by a small, explicit Lean argument. If any check fails, the certificate is rejected and the property is reported as unverified; the system never silently accepts a malformed artifact.

Certificate families. Certificates fall into two families:

- **Node-wise enclosure certificates** supply $[l_i, u_i]$ and optionally affine forms $L_i(x), U_i(x)$ per node, sufficient to discharge an output-level property directly from the propagated enclosure. IBP and CROWN/LiRPA both fall here; the difference is whether bounds are intervals or affine functions of the input.

- **Branch-and-bound leaf certificates** partition the input region B into leaves, each with its own node wise certificate. The checker validates coverage, per-leaf soundness, and that the target property holds on every leaf; global validity follows by case analysis. This handles properties no single relaxation can certify, at the cost of a larger artifact and checking time linear in the number of leaves.

One ReLU neuron under α/β-CROWN.

Let $z \in [l, u]$ be the pre-activation and $y = \mathrm{ReLU}(z)$. The phase $\beta \in \{-1, 0, 1\}$ encodes what the IBP interval tells us: provably off, provably on, or uncertain. *Inactive* (β=-1, $u \leq 0$). The interval lies entirely in the non-positive half-line, so $\mathrm{ReLU}(z) = 0$ everywhere on $[l, u]$. Both bounds are the exact zero line $(s^U, b^U) = (s^L, b^L) = (0, 0)$; no relaxation gap is introduced. *Active* (β=1, $0 \leq l$).

The interval lies entirely in the non-negative half-line, so $\mathrm{ReLU}(z) = z$ exactly. Both bounds are the identity $(s^U, b^U) = (s^L, b^L) = (1, 0)$; again, zero gap and exact propagation.

Unstable (β=0, $l < 0 < u$). The interval straddles zero and a linear relaxation is unavoidable. The upper bound is the CROWN secant connecting $(l, 0)$ to (u, u), tight at both endpoints:

$$s^U = \frac{u}{u-l}, \qquad b^U = \frac{-ul}{u-l}.$$

The lower bound is the α-CROWN-family $(s^L, b^L) = (\alpha, 0)$ with $\alpha \in [0, 1]$ chosen by the external optimizer to tighten the downstream objective. α=0 gives the trivial bound $y \geq 0$; α=1 gives the identity $y \geq z$, which is tightest when z is likely positive. Intermediate values trade local tightness against the global verification goal; in practice α is optimized by gradient ascent on the final bound, treating each neuron's slope independently.

Propagation. Affine forms compose exactly through linear layers using the $W = W^+ - W^-$ decomposition, where $W_{ij}^+ = \max(W_{ij}, 0)$ and $W_{ij}^- = \max(-W_{ij}, 0)$ route lower and upper input bounds to the correct output bound without case splits. Inactive and active neurons pass forms through without loss. Only unstable neurons accumulate gap, which shrinks as IBP intervals tighten. After processing all nodes in topological order, the output node carries certified bounds $L_{\text{out}}(x), U_{\text{out}}(x)$ over the full input box B. The checker then discharges the target property—typically $L_{\text{out},y}(x) > U_{\text{out},k}(x)$ for all $k \neq y$—via a small explicit Lean argument whose only premises are the computed scalar bounds, keeping the trusted kernel minimal.

What the Lean checker validates (conceptual checklist). Given a certificate and the IR graph:

1. **Schema + typing:** node IDs exist; shapes match; per-node bound vectors have the right dimension.
2. **Local soundness per node:** for each node i, the certified bound is implied by parent bounds and the `OpKind`-specific enclosure rule (interval rule or affine relaxation rule).
3. **Goal reduction:** the final property (e.g., robustness margin, Lyapunov decrease) is derived from the certified output enclosure by a small, explicit argument in Lean (e.g., margin check or inequality chaining).
4. **Splitting (if present):** each leaf region is within the root region; the claimed per-leaf bound is sound; and the set of leaves covers (or over-approximates) the target region according to the certificate's declared semantics.

A concrete example (robustness margin). For a classifier with logits $z(x) \in \mathbb{R}^K$ and a target label y, a standard sufficient condition for certified ℓ_∞ robustness on a region R is:

$$\underline{z_y} > \max_{k \neq y} \overline{z_k},$$

where $(\underline{z_k}, \overline{z_k})$ are sound bounds on each logit over R. A certificate can therefore supply logit bounds (from IBP/CROWN/LiRPA, optionally with splitting), and the Lean checker discharges robustness by checking the margin inequality.

Certificate schemas (illustrative). Certificate-driven checking hinges on a stable schema: the artifact names the graph and supplies node-indexed data. At a high level, a node wise enclosure certificate looks like:

```
{
  "graph_id": "...",
  "input_region": { ... },
  "bounds": {
    "node_id": { "lo": [...], "hi": [...]
    },
    ...
  }
}
```

More advanced artifacts may include affine coefficients, objectives, and branch-and-bound trees. The checker described here focuses on validating the enclosure constraints needed to discharge the target theorem, rather than replaying the optimizer's full search/parameter-optimization internally. Planned extensions (e.g., richer dual-feasibility checks for full solver internals) are described in Appendix E.3.

E.2. VNN-COMP / VNN-LIB interface (ONNX + VNNLIB)

VNN-COMP is an annual community competition designed to enable fair, objective comparison of neural network verification tools by standardizing interfaces, benchmarks, and evaluation pipelines (Brix et al., 2024; VNN-COMP, 2024). VNN-COMP instances are packaged in standardized formats: networks are provided as ONNX models and properties are specified in VNN-LIB, an SMT-LIB-style language that defines both syntax and semantics for satisfiability queries over neural networks (Brix et al., 2024; VNN-LIB, 2021). This regime is directly relevant to semantic drift: most verifiers consume exported ONNX artifacts and interpret operator semantics outside the training framework, and ONNX operator meaning is tied to explicit operator-set (opset) versioning. Our goal here is not to re-implement the entire ONNX/VNN-LIB toolchain inside Lean, but to make the conversion boundary explicit and keep the trusted core inside Lean.

Accordingly, we adopt a producer/checker workflow. A lightweight Python export step reads the ONNX network and the VNN-LIB specification and emits a compact JSON bundle containing the network structure/weights (in a typed graph form), the input region (typically an axis-aligned box), and the property matrices/constraints extracted from VNN-LIB. Lean then compiles this bundle into our operator-tagged SSA/DAG IR and checks a *sufficient UNSAT* condition by replaying IBP/CROWN style bounds against the shared IR semantics. For these suites we run the checker under the fast runtime `Float` backend (binary64), since the objective is to demonstrate a semantic benchmarking interface rather than Float32 deployment conformance.

The check we perform follows the standard sound-but-incomplete pattern used by bound-propagation verifiers: if the propagated output enclosure implies that the VNN-LIB predicate cannot hold on the entire input region, we return `safe` (UNSAT proved); otherwise we return `unknown`. Soundness is semantic: whenever Lean reports `safe`, the conclusion is derived from the Lean denotation of the compiled IR, so the trusted computing base is the IR semantics plus the small checker, with the export step treated as an explicit, auditable boundary.

Table 2 reports a small MNIST-FC slice under this interface. Lean IBP is conservative, while objective-dependent CROWN refutes a subset of properties inside Lean. Importing optimized ReLU α slopes primarily improves runtime rather than refutation count, suggesting that further tightness typically requires richer artifacts such as improved intermediate bounds and/or splitting, consistent with broader lessons emphasized in VNN-COMP reports (Brix et al., 2024).

This interface follows the same certificate/checker style as the rest of TORCHLEAN. The Lean runner replays native IBP/CROWN style bounds on the shared IR; stronger untrusted producers (e.g., optimized α/β-CROWN bounds, split certificates, or other solver artifacts) can be treated as certificate generators whose outputs are validated against the IR semantics. This gives a concrete path toward theorem-prover participation in standardized verification benchmarks by making semantics and checking the reference point, rather than re-implementing every optimizer heuristic in Lean (Brix et al., 2024; VNN-COMP, 2024).

E.3. Case Studies

Certified robustness workflow (classifier). We illustrate the full verification pipeline on *local* certified robustness for a classifier. Let $f : \mathbb{R}^d \to \mathbb{R}^K$ denote the network's *logit* function (so the predicted label is $\arg\max_k f_k(x)$), and fix a test input x_0 with nominal label $y = \arg\max_k f_k(x_0)$. Given a perturbation radius ε, we define the region of interest $R = \{x : \|x - x_0\|_\infty \le \varepsilon\}$ (or an axis-aligned box that over-approximates it). Using IBP or CROWN/LiRPA, we compute sound bounds $(\underline{z}_k, \overline{z}_k)$ such that $f_k(x) \in [\underline{z}_k, \overline{z}_k]$ for all $x \in R$. IBP provides a fast interval enclosure baseline, while CROWN/LiRPA tightens these bounds by propagating sound linear relaxations of nonlinearities and (optionally) using objective-dependent back-substitution.

Theorem E.3: Logit-margin robustness certificate

Let $f : \mathbb{R}^d \to \mathbb{R}^K$ be a classifier and R an input region. Assume we have sound logit bounds for all classes: for each k and all $x \in R$, $f_k(x) \in [\underline{z}_k, \overline{z}_k]$. If

$$\underline{z}_y \;>\; \max_{k \ne y} \overline{z}_k,$$

then $\arg\max_k f_k(x) = y$ for all $x \in R$ (i.e., the prediction is certified ℓ_∞-robust on R).

How the certificate is checked. The certificate checking step is deliberately small and semantic: the checker validates that the reported bounds are a sound enclosure of the IR semantics on R (shape-consistent, op-consistent, and region-consistent), and then applies the margin lemma above to conclude label invariance. This is exactly the "infinite-to-finite" reduction that makes bound propagation useful: instead of enumerating all $x \in R$, we certify robustness by a finite set of inequalities on the computed output enclosure. (This margin-style condition is standard in certified robustness pipelines.)

Structure of the example.

1. *Model definition:* define the classifier once in TORCHLEAN.
2. *Lowering:* lower the same program to the shared operator-tagged SSA/DAG IR (the semantic target).
3. *Bounding:* run IBP or CROWN/LiRPA on the IR to obtain logit enclosures over R.
4. *Checking:* verify the enclosure constraints and discharge robustness via Lemma E.3.

The final claim is a Lean theorem about the denotation of the shared IR, not a post-hoc interpretation of a tool output: external verifiers may *produce* bounds, but Lean *checks* that those bounds imply the semantic property of interest on the executed artifact.

Physics-Informed Neural Networks (PINNs). PINNs enforce physics by penalizing a PDE residual built from a neural field u_θ and its derivatives. We fix (i) a residual operator $\mathcal{R}$ (e.g., for Burgers/heat-type equations), (ii) a domain Ω, and (iii) a trained network u_θ, and aim to certify a uniform residual bound $\sup_{x\in\Omega} |\mathcal{R}(u_\theta)(x)| \le \varepsilon$. This is the standard PINN correctness goal: the learned model approximately satisfies the governing PDE across the domain. The verification step is derivative-dependent, so we combine (a) a proved first-order reverse-mode result that anchors the meaning of ∇u_θ to the graph semantics (under the usual smoothness/pointwise side conditions), with (b) *derivative-aware bound propagation* on the shared operator-tagged graph that produces interval enclosures for u_θ and the specific derivatives appearing in $\mathcal{R}$ (including second derivatives for the low-dimensional smooth-operator subset used in the demos). We do *not* obtain u'' by differentiating the backward-pass graph; instead, we bound the required derivatives directly via specialized bound-propagation passes on the forward graph, and then combine these enclosures to conclude the residual inequality.

Theorem E.4: Residual certificate pattern (PINNs)

Let Ω be a domain and suppose we have sound interval enclosures on Ω for every term that appears in the residual $\mathcal{R}(u_\theta)$ (e.g., bounds on u_θ, $\partial u_\theta/\partial t$, $\partial u_\theta/\partial x$, $\partial^2 u_\theta/\partial x^2$). If these enclosures imply $|\mathcal{R}(u_\theta)(x)| \le \varepsilon$ for all $x \in \Omega$, then the PDE residual of u_θ is certified within tolerance ε on Ω.

Listing 20. Schematic residual checker for a one-dimensional Burgers-style PINN.

```
structure DerivBounds where
  u    : Box scalar
  ut   : Box scalar
  ux   : Box scalar
  uxx  : Box scalar

def burgersResidual (nu : Float) (b :
    DerivBounds) : Box scalar :=
```

```
  -- R(u) = u_t + u * u_x - nu * u_xx
  b.ut + intervalMul b.u b.ux - scaleBox nu
    b.uxx

def checkResidual? (nu eps : Float) (b :
    DerivBounds) : Bool :=
  let r := burgersResidual nu b
  r.lo >= -eps && r.hi <= eps
```

The checker separates two obligations. First, the derivative bound passes produce sound boxes for u, u_t, u_x, and u_{xx} over the verification region. Second, ordinary interval arithmetic combines those boxes according to the PDE residual formula. This separation keeps the theorem statement short: any future residual operator can reuse the same pattern once its derivative boxes and algebraic interval rules are available.

Neural controller (Lyapunov-style safety/stability). We also evaluate a two stage controller-verification workflow common in learning-enabled control: Stage 1 (training/search) proposes a feedback controller $u(x)$ together with a Lyapunov candidate $V(x)$; Stage 2 (certification) proves region-based inequalities that imply safety/stability (e.g., $V(x) \geq 0$ and $\dot{V}(x) \leq -\rho(\|x\|)$ on a region). This aligns with recent neural-control verification pipelines that use bound propagation and α/β-CROWN-style tooling to certify Lyapunov conditions. In our setting, we compute enclosures for $V(x)$ and $\dot{V}(x) = \nabla V(x) \cdot f(x, u(x))$ over the region (with ∇V grounded by the same autograd semantics as execution), and the final claim is discharged by checking the resulting inequalities as a theorem about the shared IR denotation.

Theorem E.5: Lyapunov certificate pattern (region-based)

Let V be a candidate Lyapunov function for a closed-loop system $\dot{x} = f(x, u(x))$ on a region R. If certified bounds over R establish $V(x) \geq 0$ (and $V(x) = 0$ only at the equilibrium) and $\dot{V}(x) \leq 0$ (or a strict decrease condition) for all $x \in R$, then the corresponding safety/stability property holds on R under the chosen Lyapunov criterion.

Listing 21. Schematic Lyapunov inequality check over a certified region.

```
structure LyapBounds where
  Vlo    : Float
  Vhi    : Float
  VdotLo : Float
  VdotHi : Float

def checkLyapunovBox (b : LyapBounds) (rho
    : Float) : Bool :=
  -- Typical sufficient obligations on a
    box away from the equilibrium.
  (0.0 <= b.Vlo) && (b.VdotHi <= -rho)

def checkPartition (boxes : List LyapBounds
    ) (rho : Float) : Bool :=
  boxes.all (fun b => checkLyapunovBox b
    rho)
```

This is the finite reduction used by the controller case study. The analytic Lyapunov criterion quantifies over a region; the certificate partitions the region into finitely many boxes and provides bounds for V and $\dot{V}$ on each box. Lean checks the per-box inequalities and the coverage assumptions recorded in the certificate schema.

Hopfield networks and global dynamics (complementary case study). Hopfield networks are recurrent, energy-based models whose core guarantees are *global dynamical* statements: rather than certifying a local property around one input, we prove properties of entire *trajectories* generated by repeatedly applying an update rule (e.g., asynchronous coordinate updates). Classically, one defines an energy/Lyapunov functional $E(x)$ and proves that each update step does not increase E; because the state space is finite, monotone energy then implies convergence to a fixed point (an attractor) under standard symmetry and zero-diagonal conditions.

In the recent paper, Formalized Hopfield Networks and Boltzmann Machines (Cipollina et al., 2025), the authors develop a Lean 4 formalization aimed squarely at such global properties like convergence for deterministic Hopfield dynamics and ergodicity for stochastic Boltzmann machines. They emphasize that many ML formalizations mirror modern execution frameworks (sequential layers or DAG-style computational graphs), which is convenient for feedforward computation but does not directly express recurrent update semantics; because their focus is convergence/ergodicity of *recurrent* models, they adopt a graph-based dynamical-systems perspective (directed graphs/Markov kernels) integrated with mathlib/PhysLean. Their discussion highlights a real tradeoff: representations optimized for feedforward dataflow can require extra machinery (unfolding or fixed-point semantics) to model recurrence faithfully.

Our treatment is complementary: we express the Hopfield update operator and energy functional *inside the same typed tensor/program semantics* used throughout TORCHLEAN, and we prove the standard monotone-energy and convergence-style results by reasoning about an explicit state-transition map (iteration) plus a Lyapunov decrease argument. These results show that an SSA/DAG semantic core for *computation* can still support proofs about *global* dynamical behavior.

Theorem E.6: Hopfield energy decrease and convergence (informal template)

Let $x \in \{-1, +1\}^n$ be the network state, with symmetric weights $W = W^\top$ and zero diagonal, and thresholds θ. Define the energy

$$E(x) \;=\; -\tfrac{1}{2}\, x^\top W x \;+\; \theta^\top x.$$

Consider asynchronous updates that flip a single coordinate x_i according to the sign of its local field $(Wx - \theta)_i$ (with a fixed tie-handling convention). Then each update step satisfies $E(x^{t+1}) \le E(x^t)$, and hence any trajectory produced by repeatedly applying the update rule reaches a fixed point in finitely many steps (a local minimum of E under the update dynamics).

F. Universal Approximation Theorems

Universal approximation (real-valued). A universal approximation theorem (UAT) formalizes the idea that a simple network class can approximate *any* continuous function on a compact domain to arbitrary accuracy. Classic results show that single-hidden-layer networks are dense in $C(K)$ for compact $K \subseteq \mathbb{R}^d$ under mild activation assumptions (Cybenko, 1989; Hornik, 1991); more recent work characterizes approximation rates for ReLU networks (Yarotsky, 2017). In TORCHLEAN, we mechanize a standard real-valued ReLU UAT on compact domains and then specialize it to boxes such as $[-1, 1]^d$.

Theorem F.1: ReLU universal approximation on compact sets

For every compact $K \subseteq \mathbb{R}^d$, every continuous $f : K \to \mathbb{R}$, and every $\varepsilon > 0$, there exist a width m and a single-hidden-layer ReLU network N_θ such that

$$\sup_{x \in K} |N_\theta(x) - f(x)| < \varepsilon.$$

F.1. Float32 execution Soundness

Why the Float32 setting is harder. Real-valued UATs reason about exact arithmetic, whereas deployed models run under finite-precision float semantics. In this setting, even the "target function" must be interpreted carefully: the executed program computes a *rounded* function, and abstract-interpretation semantics (e.g., intervals) must account for rounding and finite-range effects. Recent work proves a floating-point analog of *interval universal approximation* (IUA), showing that floating-point networks can capture the *direct image map* of a suitably rounded target function under interval semantics (Hwang et al., 2025). In contrast, our goal in TORCHLEAN is to build a compositional, checkable foundation that supports practical verification workflows under explicit Float32 semantics, without claiming the full "exact-hull direct image" property for arbitrary networks.

Our claims. We do *not* claim a full IUA theorem of the form "the interval semantics returns the exact output-range hull for every input box" for arbitrary networks under Float32 execution. Instead, we mechanize three ingredients that are sufficient for approximation-style arguments and verification pipelines:

- **Executable Float32 semantics in the prover.** An executable IEEE-754 binary32 model (`IEEE32Exec`) that gives a precise internal meaning to "Float32 execution" (including corner cases) for full demos and checking.
- **Sound interval evaluation on a supported fragment.** A theorem-ready specification of interval semantics and a proved-sound interval evaluator for a small ReLU-MLP fragment, enabling certified enclosures needed by robustness/PINN/controller checks.
- **Error decomposition for Float32 execution approximation.** Theorem templates that bound execution error by an explicit sum of terms: (real approximation error) + (parameter/representation error, e.g. quantization) + (IEEE rounding/execution error), under explicit finiteness/no-NaN/no-Inf hypotheses.

Note: Where (Hwang et al., 2025) targets a floating-point IUA result that exactly matches the interval *direct image map* of a rounded target function, our development emphasizes compositional error bounds and sound enclosures that integrate cleanly with certificate checking and full verification workflows, without requiring the full exact-hull IUA guarantee for all networks.

Notation for the theorems below. Let $\mathbb{F}_{32}$ denote IEEE-style Float32 values (excluding NaNs/Infs when we explicitly assume *finite execution*), and let $\mathrm{toReal} : \mathbb{F}_{32} \to \mathbb{R}$ interpret a finite float as a real number. A *box* B is an axis-aligned product of Float32 intervals (a valid element of the interval domain), and $\gamma(B) \subseteq \mathbb{F}_{32}^d$ denotes its *concretization* (the set of Float32 points represented by B). We write $\nu : \mathbb{F}_{32}^d \to \mathbb{F}_{32}$ for point semantics (Float32 execution) and $\nu^\sharp : \mathsf{Box}_{32}^d \to \mathsf{Box}_{32}$ for interval semantics (abstract interpretation over boxes).

Theorem F.2: Exact-hull specification (Float32 interval semantics)

Fix $d \in \mathbb{N}$ and a NaN-free target function $\hat{f} : \mathbb{F}_{32}^d \to \mathbb{F}_{32}$. Assume a point semantics ν and an interval

semantics $\nu^\sharp$ such that $\nu^\sharp(B)$ is a box output for any valid input box B. We say $\nu^\sharp$ returns the *exact hull* of $\hat{f}$ on B if, for every valid box $B \subseteq [-1,1]^d$, there exist witnesses $m, M \in \mathbb{F}_{32}$ satisfying:

1. (**Witnessed extrema**) m and M witness the range of $\hat{f}$ on the concretization: $m \le \hat{f}(x) \le M$ for all $x \in \gamma(B)$, and there exist $x_m, x_M \in \gamma(B)$ such that $\hat{f}(x_m) = m$ and $\hat{f}(x_M) = M$.
2. (**Hull equality**) the concretization of the abstract output box is exactly the interval hull:

$$\gamma(\nu^\sharp(B)) \;=\; \{\, y \in \mathbb{F}_{32} \mid m \le y \wedge y \le M \,\}.$$

Why witnesses? This avoids committing to a particular implementation of float `min`/`max` at ties or NaN boundaries: the statement only requires existence of range witnesses and equality to the induced hull, which is the robust "Eq.(14)-style" specification used in float-IUA work.

Theorem F.3: From real approximation to Float32 execution error (1D hinge template)

Let $f : [a,b] \to \mathbb{R}$ be a real target function and let $h_\mathbb{R}$ be a hinge-style 1D network (or piecewise-linear ReLU template) parameterized by θ. Assume a real approximation bound

$$\sup_{x\in[a,b]} |f(x) - h_\mathbb{R}(x;\theta)| \le \varepsilon_{\text{approx}}.$$

Let h_{32} denote the corresponding Float32-executed program (same structure/parameters, executed in Float32) and assume *finite execution* on $[a,b]$ (no NaN/Inf and no overflow-to-Inf along the evaluation). Then for every real input $x \in [a,b]$ and any Float32 encoding x_{32} of x,

$$\big|f(x) - \mathrm{toReal}(h_{32}(x_{32};\theta))\big| \;\le\; \varepsilon_{\text{approx}} \;+\; \varepsilon_{\text{exec}}(x),$$

where $\varepsilon_{\text{exec}}(x)$ is an explicit rounding/execution envelope for this template on $[a,b]$ (e.g., a compositional bound obtained by summing per-primitive rounding terms under the finiteness hypothesis).

Theorem F.4: Three-term Float32 execution error decomposition (single-hidden-layer ReLU MLP)

Let $f : K \to \mathbb{R}$ be a target function on a domain $K \subseteq \mathbb{R}^d$ and let $N_\mathbb{R}(\cdot;\theta_\mathbb{R})$ be a real single-hidden-layer ReLU MLP. Let $N_{32}(\cdot;\theta_{32})$ be its Float32-executed counterpart with Float32 parameters θ_{32}, and let $\theta'_\mathbb{R} := \mathrm{toReal}(\theta_{32})$ be the real interpretation of those float parameters. Assume:

1. (**Real approximation**)

$$\sup_{x\in K} |f(x) - N_\mathbb{R}(x;\theta_\mathbb{R})| \le \varepsilon_{\text{UAT}}$$

2. (**Parameter/representation gap**)

$$\sup_{x\in K} |N_\mathbb{R}(x;\theta_\mathbb{R}) - N_\mathbb{R}(x;\theta'_\mathbb{R})| \le \varepsilon_{\text{param}}$$

3. (**Float32 execution gap**) for all admissible inputs (under a finiteness hypothesis),

$$|N_\mathbb{R}(x;\theta'_\mathbb{R}) - \mathrm{toReal}(N_{32}(x_{32};\theta_{32}))| \le \varepsilon_{\text{exec}}$$

Then for all $x \in K$ (with any Float32 encoding x_{32}), where any input-encoding/quantization error can be bounded separately and added to the right-hand side (or folded into $\varepsilon_{\text{exec}}$),

$$\big|f(x) - \mathrm{toReal}(N_{32}(x_{32};\theta_{32}))\big| \;\le\; \varepsilon_{\text{UAT}} + \varepsilon_{\text{param}} + \varepsilon_{\text{exec}}$$

G. Limitations and Discussion

Limitations and near-term roadmap. Execution vs. training at scale. TORCHLEAN is semantic infrastructure, not a throughput-optimized training stack. Lean execution and the optional CUDA backend prioritize a machine-checked semantic link between (i) the program users write, (ii) the IR graph we reason about, and (iii) the artifacts we check, rather than matching industrial training throughput. The CUDA path accelerates selected kernels and examples behind explicit native runtime contracts; large-scale training can also be performed externally and imported as weights/structure, with Lean used for semantic checking, certificate validation, and proof-carrying artifacts. Scaling the native backend, kernel fusion, and internal bound optimization remains an engineering extension of the same semantic design.

Verification scope and certificates. Our native verification layer covers a curated IR fragment with a proved-sound IBP core and a CROWN/LiRPA style affine engine, together with certificate checking infrastructure for externally produced bound artifacts. We support an α/β-CROWN certificate dialect for the graph-based verifier: certificates may supply IBP pre-activation boxes, affine bounds, α parameters for unstable-ReLU lower relaxations, and optional β phase vectors that are checked for consistency and then replayed by the Lean step semantics. When strongest current tightness is required beyond this interface, we rely on external optimizers as *untrusted producers* and check their exported bounds/leaf certificates against the shared IR semantics, keeping the trusted computing base to

the Lean checker plus the IR denotation. Extending the checker to validate additional solver families (e.g., richer dual-feasibility conditions, cutting-plane certificates, or SDP-based relaxations) is left to future extensions; Appendix E and Appendix E.3 document the present schemas and extension points.

Float32 trust boundary (the "hardware gap"). We separate (i) real-valued reference semantics, (ii) proof level rounding-on-$\mathbb{R}$ models (FP32/NF) for compositional error envelopes, and (iii) an executable bit-level Float32 kernel (`IEEE32Exec`). What remains is a *target-specific* refinement from a deployed `Float32` toolchain to the executable model: one must fix a deployment configuration (rounding mode, denormal policy, contraction/ reassociation, and reduction order constraints) and then discharge a conformance interface (by proof or explicit validation). Appendix D.1 summarizes the internal refinements we prove and the remaining target-level obligations.

Deployment and code generation. Inside Lean, we close the training and verification semantic gap by making the operator-tagged IR the single semantic target. Deploying a verified model on an embedded target still introduces a translation step: either run a small IR interpreter on-device, or generate C/Rust code from the IR and connect its behavior back to the Lean denotation. Lean's compilation pipeline and runtime already support efficient compiled code, and there are emerging toolchains that bridge verified Lean reasoning with real systems languages (e.g., Rust-to-Lean verification pipelines), suggesting concrete paths toward deployment pipelines.