Verified Deep Learning with Lean 4

brett koonce

Version 0.5.3

Contents

1 Introduction **2**

2 You Are Here **5**

3 MLP: Dense Layer **10**
3.1 Example: MNIST MLP . . . 10
3.2 What's inside `.train`? . . . 13

4 CNN: Convolution and Pooling **16**
4.1 Example: MNIST 2D CNN . . . 16

5 BatchNorm: the hard one **19**
5.1 Example: the BN lift on CIFAR . . . 19

6 Residual: Skip Connections **22**
6.1 Example: ResNet-34 on Imagenette . . . 22
6.2 Ablation: what each ingredient contributes . . . 24

7 Depthwise Convolution **26**
7.1 Example: MobileNet V2 on Imagenette . . . 26

8 Squeeze-and-Excitation **29**
8.1 Example: EfficientNet-B0 on Imagenette . . . 29

9 LayerNorm and GELU **32**

10 Attention and the ViT finale **33**
10.1 Example: ViT-Tiny on Imagenette . . . 34

11 Bestiary of Architectures **36**
11.1 Bestiary-only `Layer` primitives . . . 36
11.2 Bestiary entries . . . 38
11.2.1 Vision classifiers — Part 1's primitives at scale . . . 38
11.2.2 Object detection . . . 42
11.2.3 Semantic segmentation . . . 44
11.2.4 Image generation . . . 46
11.2.5 Reinforcement learning . . . 48
11.2.6 Beyond vision . . . 49

A Getting started **54**

B On Verification **57**
B.1 Axioms . . . 57
B.2 Finite-difference gradient checks . . . 58
B.3 The JAX parallel pipeline . . . 58

Chapter 1

Introduction

On the next-to-last page of my first book, I wrote that I dreamed of a future where we could take our code and compile it for whatever backend we desire, via MLIR. I meant it as a wishlist item — something I expected to happen eventually, but not soon enough to matter for that work.

It turned out "eventually" was about five years.

This is the book I wanted to write back then, and couldn't. The tools weren't ready. Lean 4 was still in development. MLIR was a research project. IREE couldn't compile a training step. Now all three are production-quality, and the thing I wished for is not only possible — it's sitting in a GitHub repo, training ResNets on a single GPU, with machine-checked proofs that every gradient is correct.

What this book is about

This book teaches you how backpropagation actually works.

Not the hand-wave version — "compute gradients, update weights, repeat." You already know that version. If you've trained a model, you've called `.backward()` a thousand times. What you probably haven't done is open the hood and look at what `.backward()` is actually computing, layer by layer, and verify that the computation is mathematically correct.

That's what we're going to do. For every layer in the modern stack — dense, ReLU, convolution, pooling, batch normalization, residual connections, depthwise convolution, squeeze-and-excitation, layer normalization, and self-attention — we will:

1. **Derive the backward pass** from the forward pass, using nothing but the chain rule and some index algebra.
2. **State the result formally** in Lean 4, a programming language that doubles as a theorem prover.
3. **Show the MLIR** that compiles to GPU bytecode, and verify it matches the math.
4. **Train the model** on real data, end-to-end, on your hardware.

Three views of the same computation: the math, the proof, the code that runs. They agree, and the entire book is about understanding why.

The thesis

Here's the punchline, stated up front so you know where we're headed.

Backpropagation is **one operation** (the vector-Jacobian product), composed using **three rules**:

1. **The chain rule** — if you compose two functions, you compose their backward passes.
2. **Additive fan-in** — if two paths feed into a sum (like a residual connection), their gradients add.
3. **Multiplicative fan-in** — if two paths feed into a product (like squeeze-and-excitation or attention), the product rule gives you two backward contributions that add.

And **five structural tricks** for layers whose Jacobians are dense but exploitably structured:

1. **Diagonal** — elementwise activations (ReLU, GELU, Swish). The Jacobian is diagonal; the backward is one multiply.

2. **Sparse Toeplitz** — convolutions. The Jacobian is block-sparse with a sliding-window structure; the backward is another convolution with a reversed kernel.

3. **Binary selection** — max pooling. The Jacobian routes the gradient to whichever input was the max.

4. **Rank-1 correction to diagonal** — softmax, batch norm, layer norm. The Jacobian is dense but has a clean closed form because it's a diagonal plus a rank-1 outer product.

5. **Outer product reductions** — dense layers and matmul. The weight gradient is the outer product of the input with the output gradient.

That's it. Three rules, five tricks. Every architecture in this book — MLP, CNN, ResNet, MobileNet, EfficientNet, Vision Transformer — and every architecture in the bestiary — U-Net, DETR, Mamba, GAN, VAE, diffusion models, CLIP, GPT-style decoders — decomposes into some combination of these eight things. There is no ninth.

I didn't set out to prove this. I set out to train a ResNet with verified gradients, and by the time I'd done it for enough architectures, the pattern became impossible to ignore. The framework kept working without needing new primitives. Once you see it, you can't unsee it, and the rest of the book is about making sure you see it.

Who this book is for

This book assumes you've trained a neural network. Not a fancy one. If you've fine-tuned a pretrained model, debugged a NaN loss, or watched a model overfit and wondered why, you're the audience.

I'm going to assume you know what a derivative is, what a matrix is, and roughly what a convolutional layer does. I'm not going to assume you know Lean, MLIR, formal verification, or compiler theory. The Lean code appears throughout the book as a precise specification and a credibility instrument — not as a barrier. If you skip every Lean snippet, you still get a complete book on the mechanics of backpropagation. The Lean is there for the reader who wants to *check* me, not the reader who needs to learn formal methods.

This is not a beginner's book. If you're looking for your first introduction to neural networks, there are several good ones: my first book is one option, fast.ai's free courses are another, and there are good resources all over the web. Come back to this one when you've trained something and want to understand what you did.

This is also not a research textbook. I'm not trying to write Goodfellow or Bishop. The proofs here are for credibility, not for theoretical depth. If you want the full formal story, Mathlib is open-source and you can extend it. What this book gives you that a research textbook doesn't is the bridge from the math to the actual GPU code, and the assurance that they agree.

If you're me from ten years ago — interested in ML, willing to put in the work, not sure where the math stops and the folklore begins — this is the book I wish I'd had.

Why now

Three things had to happen before this book was possible, and all three happened in the last few years:

Lean 4 reached maturity. Lean is a programming language that is also a theorem prover. You can write a function in Lean and then write a proof that the function does what you claim. If the proof compiles, the claim is correct — not because you trust the author, but because the compiler checked it. Lean 4 (released 2023) is the version that made this practical for non-specialists: it has a real package manager, real tooling, a growing math library (Mathlib), and an active community.

MLIR and StableHLO stabilized. MLIR is a compiler infrastructure for machine learning. StableHLO is a portable operation set built on MLIR that describes the computations neural networks perform. Together, they let you specify a training step as a sequence of mathematical operations and compile it to any hardware backend — CPU, CUDA, ROCm — without writing backend-specific code. This is the layer that turns "math on paper" into "code on the GPU."

IREE became production-ready. IREE (Intermediate Representation Execution Environment) is the runtime that takes a compiled StableHLO module and executes it on actual hardware. It handles memory management, device allocation, and kernel dispatch. It's what lets us go from a Lean spec file to a running training loop with no Python anywhere in the pipeline.

The combination of these three tools means, for the first time, that a single person can: (a) write a neural network spec in a language with a type system strong enough to state and check mathematical properties; (b) compile that spec to GPU-runnable code via a standard intermediate representation; and (c) train the network end-to-end on consumer hardware, with formal guarantees about the gradient computation. That's the pipeline this book is built around.

Why image recognition

Same reason as the first book: it's the oldest, most well-understood corner of deep learning, which means we can introduce primitives one at a time in a logical order.

We start with MNIST (digits, 28×28 grayscale) and work up through CIFAR-10 (color, 32×32) to Imagenette (real photos, 224×224). The architectures form a natural progression: MLP $\rightarrow$ CNN $\rightarrow$ ResNet $\rightarrow$ MobileNet $\rightarrow$ EfficientNet $\rightarrow$ ViT. Each one adds exactly one new structural primitive to the framework. By the time you reach ViT, you've seen every primitive there is to see for modern vision — and as it turns out, for modern language models too, since transformers are the same architecture in both domains.

The bestiary at the end demonstrates that the same primitives cover detection (DETR), segmentation (U-Net), generation (diffusion models), self-supervised learning (MAE, CLIP), sequence modeling (Mamba), and multi-modal models (LLaVA). Image recognition is the on-ramp; the destination is the whole modern stack.

Why Lean

I considered several options for the "formal" half of the book — Rocq, Agda, Isabelle, or just doing the proofs on paper with LaTeX. I chose Lean for a practical reason: it reads like code.

If you've trained a model in PyTorch, you can read:

```
def dense {m n : Nat} (W : Mat m n) (b : Vec n) (x : Vec m) : Vec n :=
  fun j => finSum m (fun i => x i * W i j) + b j
```

and understand it. It's a function with typed arguments. `{m n : Nat}` are the dimensions. `fun j =>` is a lambda. `finSum` is a for loop. You don't need to learn a new paradigm — you need to learn maybe twenty keywords and some notation, and then Lean reads like any other typed functional language.

The other reason is that Lean enforces honesty. If I claim that the backward pass of batch normalization is a specific three-term formula, the Lean compiler requires me to prove it. If the proof compiles, the claim is correct — not because you trust me, but because the type checker verified it. There is no hidden hand-waving. Every VJP in this book — dense, convolution, batch norm, residual, attention, all of them — builds with zero `sorry`s. The compiler is the referee.

Let's go

In 2018, I was on a small team that won Stanford's DAWNBench competition — training ImageNet in 3 hours for $25 on commodity hardware. The lesson from that experience was simple: most of the apparent difficulty in modern deep learning isn't fundamental. It's accumulated folklore, missing tools, and a culture that takes the math on faith.

This book applies that same lesson to the math itself. The folklore says backpropagation is complicated. It isn't. It's three rules, five tricks, and a lot of bookkeeping. The tools to verify this claim finally exist, and you're holding the result.

Three of my favorite citations of the first book are from high school students. I don't think it's an accident teenagers are publishing results. The barriers to doing real ML research used to be infrastructure — compute, data, institutional access. Those barriers have mostly fallen. What remains is the smaller barrier of understanding what you're doing at the level where you can verify it, not just run it.

This book is for the person who wants to cross that barrier. It might be you. It might be the next kid who picks it up in a library somewhere and decides to see how far they can go.

Let's find out.

Chapter 2

You Are Here

A VJP (vector-Jacobian product) is the backward function for a layer: takes an upstream gradient, returns the downstream one. Every backward function in this book is a VJP built by composing the axioms here — the partial-derivative function pdiv, its three structural rules (chain, sum, product), and three VJP record types (HasVJP, HasVJPMat, HasVJP3, one per tensor rank) that bundle a backward function with its correctness claim.

Definition 1 (Partial derivative)**.** The partial derivative function. For $f : \mathbb{R}^m \to \mathbb{R}^n$, pdiv $f\, x\, i\, j$ is the (i, j)-entry of the Jacobian at x, defined as fderiv$_{\mathbb{R}}$ $f\, x\, (\mathbf{e}_i)\, j$ — the j-th coordinate of Mathlib's Fréchet derivative applied to the i-th standard basis vector.

Theorem 2 (Chain rule)**.** pdiv$(g \circ f)\, x\, i\, k = \sum_j$ pdiv $f\, x\, i\, j \cdot$ pdiv $g\, (f\, x)\, j\, k$*, conditional on f differentiable at x and g differentiable at $f(x)$.*

Theorem 3 (Sum rule)**.** *Linearity of the derivative; conditional on both summands differentiable at x.*

Theorem 4 (Product rule)**.** *Pointwise Leibniz rule for elementwise products; conditional on both factors differentiable at x.*

Theorem 5 (Identity Jacobian)**.** pdiv(id) $x\, i\, j = \delta_{ij}$.

Theorem 6 (Constant has zero Jacobian)**.**

Theorem 7 (Gather / reindex Jacobian)**.** *Covers permutations, reshapes, slicing. Generalizes* `pdiv_id`*.*

Theorem 8 (Row-independence for matrices)**.** *Applying a vector function per-row to a matrix has block-diagonal Jacobian. Proved from* fderiv *via the row-projection* ContinuousLinearMap *and the chain rule, given a* Differentiable *hypothesis on the per-row function.*

Theorem 9 (Finite-sum rule)**.** *Linearity extended to arbitrary finite sums by induction.*

Theorem 10 (VJP chain rule)**.** *Given* HasVJP f *and* HasVJP g*, get* HasVJP $(g \circ f)$.

Theorem 11 (Additive fan-in VJP)**.** *VJP of $f + g$. Used for residual connections.*

Theorem 12 (Multiplicative fan-in VJP)**.** *VJP of elementwise product. Used for Squeeze-and-Excitation.*

Theorem 13 (Identity VJP)**.**

Theorem 14 (Matrix-level chain rule)**.** *Lifts* `vjp_comp` *via the* Mat.flatten *bijection.*

Theorem 15 (Matrix-level additive fan-in)**.**

Theorem 16 (Matrix-level identity)**.**

Theorem 17 (Scalar-scale Jacobian)**.** *Phase 6 derivation.*

Theorem 18 (Transpose Jacobian)**.**

Theorem 19 (Matmul Jacobian, left factor fixed)**.** *Phase 6: derived, not axiomatized.*

Theorem 20 (Matmul Jacobian, right factor fixed)**.**

Theorem 21 (Matmul VJP, left factor fixed)**.**

Theorem 22 (Matmul VJP, right factor fixed)**.**

Theorem 23 (Scalar-scale VJP)**.**

Theorem 24 (Transpose VJP)**.**

Theorem 25 (Row-wise lifting of any HasVJP)**.** *Phase 8 generic helper: lifts any* HasVJP $(g : \mathbb{R}^n \to \mathbb{R}^p)$ *to a* HasVJPMat *on* $\mathbb{R}^{m\times n} \to \mathbb{R}^{m\times p}$*.*

Theorem 26 (3D chain rule)**.**

Theorem 27 (3D VJP chain rule)**.**

Theorem 28 (3D additive fan-in)**.**

How to read this book

Every chapter after this one follows the same pattern:

1. A new architecture (the "what")
2. A new structural primitive (the "why")
3. Derivation of the backward pass (by hand, with indices)
4. Lean proof (the formal statement)
5. MLIR snippet (the GPU code)
6. Training results (proof it works)

You can read straight through, or you can skip to any chapter whose architecture interests you — each one is mostly self-contained, with back-references to earlier chapters for the primitives it builds on.

The Lean code and MLIR snippets are available in the lean4-mlir repository on GitHub. Every architecture in the book can be trained on your hardware. Appendix A walks you through the toolchain setup; if you're impatient, there's a Docker image that gets you from zero to training MNIST in one command.

How this book is organized

Part 1: The framework (Chapters 2–10). One new primitive per chapter, proved correct as we go. The first half pins down the composition rules — chain, additive fan-in, product — on dense layers, convolution, and normalization. The second half climbs the ladder to ViT.

- **Chapter 2: You Are Here** — Chain rule, sum rule, product rule for partial derivatives. Every backward function in the book is a VJP (vector-Jacobian product) built by composing these.
- **Chapter 3: MNIST 1D MLP** — Dense layers, ReLU, softmax CE, the chain rule. The most important chapter: everything after is adding more layers.
- **Chapter 4: MNIST 2D CNN** — Convolution, max pooling. The backward pass of conv is another conv with a reversed kernel.
- **Chapter 5: CIFAR with BatchNorm** — The first dense Jacobian with a clean closed form. Three terms, one cancellation, and you never have to derive it again.
- **Chapter 6: ResNet-34** — Residual connections. Gradients from two paths add at the input.
- **Chapter 7: MobileNetV2** — Depthwise convolution. Same as regular conv, but diagonal in the channel dimension.
- **Chapter 8: EfficientNet** — Squeeze-and-excitation. The product rule: main path times gate, gradients from both paths add.
- **Chapter 9: LayerNorm and GELU** — Batchnorm in disguise. Activation functions.

- **Chapter 10: Vision Transformer** — Self-attention. Softmax in the middle of the network, three-way fan-in at the input. The capstone.

Part 2: The bestiary (Chapter 11). A catalog of ∼37 additional architectures, from U-Net to Mamba to diffusion models, each decomposed into the framework's primitives. Organized by the question you're asking: "how do I do detection?", "how do I do generation?", etc.

Appendices. Toolchain setup (A) and the complete proof framework as a reference (B).

Foundation: Mathlib's `fderiv`

Earlier drafts of this book axiomatized the entire calculus foundation — chain rule, sum rule, product rule — as eight opaque facts. The current foundation *flips* that: pdiv is *defined* in terms of Mathlib's Fréchet derivative fderiv, and the structural rules above are theorems proved from Mathlib's API. Each carries a Differentiable hypothesis, threaded through every downstream chapter.

Many later-chapter axioms have been pruned the same way. Where earlier drafts axiomatized `conv2d` as an opaque function with a stated Jacobian, the current version defines it concretely and proves the weight/bias-gradient lemmas from the foundation. Twenty axioms survive book-wide — mostly composition-level shortcuts and hard-to-derive primitives (multi-head SDPA, the three-term BN cancellation) where the verification effort exceeded the value.

The cost is mild: Differentiable hypotheses propagate. The benefit is that every claim is either a definition Lean unfolds, a theorem typechecked against Mathlib, or one of the small remaining axioms (audited in Appendix B).

Worked example: the dense layer

The axioms above look abstract on their own. Here's how they compose into a concrete VJP, using just the tools of this chapter.

The dense layer forward is $f(x) = W \cdot x + b$, mapping $\mathbb{R}^n$ to $\mathbb{R}^m$. Its Jacobian at any x is precisely W — Theorem 29 (`pdiv_dense`, proved from the foundation rules) states this directly:

$$\text{pdiv}\ f\, x\, i\, j = W_{i,j}.$$

Step 1: package f as a HasVJP. A HasVJP record bundles a backward function with a correctness claim. For dense, the backward function is $\text{backward}(dy) = W^T \cdot dy$: multiply the upstream gradient by the transposed weights. The correctness claim $\text{backward}(dy)_j = \sum_i dy_i \cdot \text{pdiv}\ f\, x\, i\, j$ follows from pdiv *_dense* in two lines. Call the resulting record `dense_has_vjp` (proved as § 33 in the next chapter).

Step 2: compose two dense layers. Let $g(y) = W_2 \cdot y + b_2$ be a second dense layer, mapping $\mathbb{R}^m$ to $\mathbb{R}^p$. The composition $g \circ f$ maps $\mathbb{R}^n$ all the way to $\mathbb{R}^p$.

By Theorem 10 — the *VJP chain rule*, proved above from pdiv *_comp* — if f and g each have a HasVJP, so does $g \circ f$, with backward function $dz \mapsto \text{backward}_f(\text{backward}_g(dz))$.

That's it. Two applications of `dense_has_vjp` plus one application of `vjp_comp`, and a two-layer dense stack has a machine-checked backward pass.

Every later chapter follows this pattern. Chapter 3's MLP is three dense layers plus activations — the above, with `relu_has_vjp` inserted between each pair. Chapter 4's CNN replaces dense with `conv2d_has_vjp3` but the composition story is unchanged. By Chapter 10 the pattern is identical: HasVJPs for bundled primitives (conv, BN, attention) composed with the chain, fan-in, and product rules from this chapter.

One pattern, scaled up.

Axiom and theorem budget per chapter

If you're planning how much of the book to read, here's the per-chapter cost in both directions: new axioms (unproved starting facts we take as given) and new theorems (proven compositions Lean's typechecker verified). The totals across Chapters 2–10 are **81 numbered items (10 axioms + 65 theorems + 6 definitions)**, with zero `sorry`s. (Earlier drafts shipped 30 axioms; the foundation flip described above plus the follow-up diff-threading work shed 20 of them by promoting calculus and chapter-specific Jacobians from axioms to theorems and replacing several opaque forwards with concrete definitions.)

This chapter (Ch 2) does the structural heavy lifting: zero axioms and 27 theorems that compose Mathlib's fderiv into the VJP toolkit every later chapter will reuse — the chain rule, additive and multiplicative fan-in, identity, reindexing, row-independence for matrices, and the matrix-/ 3-tensor-level lifts. Plus pdiv itself as a definition over fderiv.

Chapter 3 (MLP) adds 3 axioms (the guarded ReLU subgradient plus two composition shortcuts — `relu_has_vjp` and `mlp_has_vjp` — that bundle existence at non-smooth points) and 6 theorems. Dense Jacobians and the softmax cross-entropy gradient are now proved from the foundation rather than axiomatized.

Chapter 4 (CNN) adds 2 axioms (`conv2d` and `maxPool2` input VJPs) and 2 theorems plus 2 definitions. `conv2d` and `maxPool2` are now concrete defs; their weight- and bias-gradient lemmas are proved from the foundation; only the input-side VJP remains opaque (padding boundary / argmax routing).

Chapter 5 (BN) adds zero axioms and 7 theorems. The inverse-stddev broadcast Jacobian and its smoothness lemma under $\varepsilon > 0$ are now proved from foundation, as are the BN affine and centering Jacobians.

Chapter 6 (Residual) adds zero new axioms and just 1 theorem (the residual additive-fan-in VJP, built entirely from Chapter 2's kit). That's the scaling claim in miniature — new architecture, no new calculus.

Chapter 7 (Depthwise) adds 1 axiom (depthwise input VJP) and 2 theorems plus 1 definition. Same pattern as Ch 4: the forward is a concrete def, weight/bias gradients are proved from foundation, only the input-side VJP is left opaque.

Chapter 8 (SE) adds zero axioms and 1 theorem (`seBlock_has_vjp`). Squeeze-and-excitation is global-avg-pool plus two 1×1 convs plus the elementwise product; every piece was already proved elsewhere.

Chapter 9 (LN + GELU) adds zero axioms and 3 theorems plus 2 definitions. The GELU Jacobian is proved via fderiv and `Real.differentiable_tanh`; `geluScalar` and its derivative are concrete defs; LayerNorm reuses the BN proof template.

Chapter 10 (Attention) adds 4 axioms and 16 theorems. Two of the axioms are bundled multi-head SDPA — the VJP itself and its Differentiable sibling — where deriving from matmul and rowSoftmax would require per-head reduction machinery beyond the chapter's scope. The other two cover the ViT patch-embedding interface (VJP + Diff sibling), an opaque-codegen boundary. Softmax Jacobian, row-wise softmax smoothness, and seven transformer-level chains (sublayers, block, tower, ViT body) are all now proved. The theorems call the matrix-composition kit Chapter 2 set up; the transformer block factors through every one of them.

Chapter	New (ax + th + def)	Cumulative	Last #	Architectures unlocked
2 (You Are Here)	0 + 27 + 1	0 + 27 + 1	28	foundation
3 (MLP)	3 + 6 + 0	3 + 33 + 1	37	MLP
4 (CNN)	2 + 2 + 2	5 + 35 + 3	43	+ CNN
5 (BN)	0 + 7 + 0	5 + 42 + 3	50	+ BN-nets
6 (Residual)	0 + 1 + 0	5 + 43 + 3	51	+ ResNet-34
7 (Depthwise)	1 + 2 + 1	6 + 45 + 4	55	+ MobileNet V2
8 (SE)	0 + 1 + 0	6 + 46 + 4	56	+ EfficientNet
9 (LN + GELU)	0 + 3 + 2	6 + 49 + 6	61	+ transformer infra
10 (Attention)	4 + 16 + 0	10 + 65 + 6	81	+ ViT

The "Last #" column is the highest LaTeX-rendered item number at the end of each chapter (axioms, theorems, and definitions share the counter), so a reader who sees "Theorem 81" near the end of Ch 10 can look here and confirm where the count came from.

Two observations worth pulling out. First, **five chapters (2, 5, 6, 8, 9) add zero axioms**: every primitive in those chapters either inherits from Mathlib's fderiv or composes over previously-proved theorems. That's the scaling claim the book is making — new architectures, no new calculus — in its cleanest form. Second, **the only axioms left are at the boundaries of smoothness or codegen**: the guarded ReLU subgradient and its two existence shortcuts in Chapter 3, the input-side VJPs of `conv2d` / `maxPool2` / `depthwiseConv2d` where padding boundary and argmax routing complicate symbolic differentiation, bundled multi-head SDPA, and the opaque ViT patch-embedding interface in Chapter 10. Each is a deliberate "stays axiomatic" boundary, not a deferred proof.

Roadmap: skip to your target architecture

Every theorem and axiom in the book carries a `\uses{}` annotation, so the dependency graph from "which target architecture do I care about?" back to "which Ch 2 items do I actually need?" is explicit — we get it for free from doing the proofs in Lean in the first place. If your goal is a specific architecture rather than the full tour, here's the critical-path subset of this chapter per target.

Target: MLP (Ch 3). Chain rule plus a handful of basic helpers: `vjp_comp`, `flatten`, `unflatten`, `mulVec`, `outer`. About a third of this chapter. Skip every 2D-matrix theorem (the `*Mat`-named ones) and every 3D-tensor theorem (the `*3`-named ones).

Target: CNN (Ch 4), with or without BN (Ch 5). MLP subset plus the 3D-tensor composition tools: `pdiv3`, `pdiv3_comp`, `vjp3_comp`, `flatten_unflatten`, `unflatten_flatten`, `transpose`. Still no matrix composition. About half of this chapter. BN itself is self-contained in Ch 5's own axioms; it only reuses `vjp_comp` from here.

Target: ResNet-34 (Ch 6). CNN subset plus `biPath_has_vjp` and `identity_has_vjp` for the additive skip connection. *This is a great stopping point.* ResNet-34 was state of the art for years, still routinely gets 90% on Imagenette; its slightly bigger brother ResNet-50 is the standard benchmark ( ) for production image classification on consumer hardware, and its proofs don't need anything more complicated than the chain rule and two additive-fan-in lemmas. If your destination is a real image-classifier that works, Ch 2 first-half plus Ch 3-6 is a complete reading list; Ch 7–10 are strictly skippable.

Target: MobileNet V2 (Ch 7). ResNet subset unchanged. The depthwise chapter introduces its own axioms but doesn't add any Ch 2 dependencies — the tensor-calculus tools stay the same as for ResNet.

Target: EfficientNet (Ch 8). MobileNet subset plus `elemwiseProduct_has_vjp` for the SE gate. Still no matrix composition.

Target: LayerNorm + GELU (Ch 9). Same Ch 2 subset as EfficientNet. Ch 9 is self-contained in its own axioms and doesn't reach back into the tensor-calculus chapter at all.

Target: Attention / ViT (Ch 10). You finally need the 2D-matrix half of this chapter: the full `pdivMat` / `vjpMat_comp` / `biPathMat_has_vjp` / `identityMat_has_vjp` composition kit, plus `transpose_has_vjp`, `matmul_left_const_has_vjp`, `matmul_right_const_has_vjp`, `scalarScale_has_vjp`, `rowwise_has_vjp_mat`, and the `hasVJPMat_to_hasVJP` bridge back to flat HasVJP-land. This is why the matrix half of Ch 2 exists; every earlier chapter routes around it.

If you're unsure: read the first half of this chapter carefully (tensor helpers, chain rule, additive fan-in) and skim the matrix second half. You can always come back for it when you're ready to tackle attention.

For readers of the first book

If you're coming to this book from *Convolutional Neural Networks with Swift for TensorFlow*, you already know most of the architectures and the forward-pass intuition: VGG, ResNet, MobileNet, and EfficientNet are all familiar, and you've built and trained them at least once.

Chapters 9 and 10 will be new material even on the forward-pass side. I shipped the final draft of the first book in July 2020; in late August the grapevine started whispering that something attention-shaped was coming, and the ViT paper landed on arXiv that October. Everything in this book that looks like a transformer — LayerNorm, GELU, multi-head attention — postdates the first book and will be genuinely new for you. Otherwise, this book is the same architectures brought under the additional scrutiny of a formal backward pass.

Concretely, that means:

- **The architecture introductions in chapters 3-8 will feel redundant.** You can skim past the forward-pass descriptions — they're first-book content, kept here so the chapters stay self-contained — and head straight for the `*_has_vjp` theorems and the `\uses{}` annotations that pin down the backward pass.
- **Chapter 2 is the new material for you.** The tensor-calculus chapter is what the first book didn't have — where the gradients get formalized. If you only read one chapter thoroughly, make it this one.
- **Training recipe upgraded.** The first book's codegen trained with SGD (momentum 0.9, learning rate 0.002) and no other tricks — one recipe held constant across architectures for consistency. This book adopts the modern recipe — Adam, cosine decay, linear warmup, weight decay, label smoothing, and basic data augmentation — which is how the accuracy numbers in this book's results table were achieved. The full recipe and each component's contribution are detailed in Chapter 6 (ResNet), where the ablation comparisons live.
- **The codegen is new.** The first book's Swift-for-TensorFlow pipeline is replaced with Lean 4 → StableHLO MLIR → IREE → GPU. Mostly doesn't affect the math; it changes where the gradients are computed (at codegen time in Lean, not at runtime by a framework).

Practical reading style: work through Chapter 2 slowly, then skim the rest at the pace of "forward pass I know → new `*_has_vjp` theorem → see what axioms it cites → move on." The dependency DAG is your friend.

Chapter 3

MLP: Dense Layer

The workhorse: $y = Wx + b$ plus ReLU and softmax cross-entropy loss.

Theorem 29 (Dense Jacobian wrt input)**.** $\partial(Wx+b)_j/\partial x_i = W_{ij}$. *Derived from foundation rules.*

Theorem 30 (Dense Jacobian wrt weight)**.** $\partial(Wx+b)_j/\partial W_{i'j'} = x_{i'}\delta_{jj'}$. *Phase 7; derived from foundation rules over the flatten bijection.*

Axiom 31 (ReLU Jacobian (guarded subgradient))**.** Pdiv of ReLU at smooth points ($\forall k,\ x_k \neq 0$). Guarded with the smoothness hypothesis after the foundation flip: the unguarded form is provably inconsistent with fderiv's junk-default at non-smooth multi-D points.

Theorem 32 (Softmax cross-entropy gradient)**.** $\partial L/\partial z = \mathrm{softmax}(z) - \mathrm{onehot}(y)$. *Proved by chain rule on* $-\log \circ \mathrm{softmax\_label}$ *using* `HasFDerivAt.log` *(with* $\mathrm{softmax}(z)[\ell] > 0$ *from exp positivity) composed with the proved* softmax *Jacobian.*

Theorem 33 (Dense VJP)**.**

Theorem 34 (Dense weight gradient is the outer product)**.** $dW = x \otimes dy$. *Phase 7 promoted from vacuous* `rfl` *to theorem.*

Theorem 35 (Dense bias gradient is identity)**.** $db = dy$. *Phase 7: derived, no new axiom.*

Axiom 36 (ReLU VJP)**.** Axiomatized after the foundation flip: with guarded `pdiv_relu`, the existence of a backward at non-smooth points becomes a convention (the standard ML subgradient routing) rather than a theorem.

Axiom 37 (MLP composition VJP)**.** Axiomatized: the chain composes through ReLU which is non-differentiable at the kink, so `vjp_comp` (which now requires Differentiable evidence) cannot mechanically compose this. Subgradient routing is the source of axiomaticness.

3.1 Example: MNIST MLP

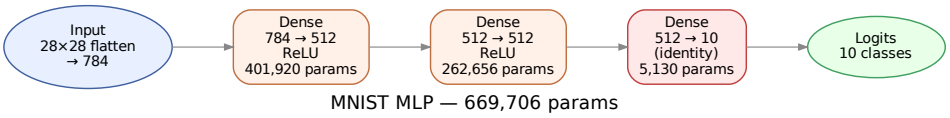
The theorems above are the calculus. Here is a concrete architecture built from those pieces: a three-layer fully-connected classifier for 28×28 MNIST digits.

Dataset overview

MNIST is the standard testbed for image-recognition learning. It's a collection of 28×28 grayscale images of handwritten digits 0–9: 60 000 training images and 10 000 test images, each with a label indicating which digit was drawn. The dataset has been around since 1998, and every textbook starts here for the same reason this one does — at 784 pixels per image and 10 classes, MNIST is small enough that you can train a competitive model on a laptop CPU in minutes while still having a nontrivial learning problem.

Our goal in this chapter is to correctly classify a held-out test digit based on a model trained from the 60 000 training digits. We're going to ignore the 2D spatial structure of the image entirely for now — just flatten each 28×28 image into a 784-dim vector and treat it as a plain supervised-learning classification problem. This is the **multilayer perceptron** (MLP). Chapter 4 revisits MNIST with convolutions that respect the spatial structure.

Architecture



Three dense layers stacked with ReLU non-linearities between them: $784 \to 512 \to 512 \to 10$. First layer ingests the flattened image. The two hidden layers let the network learn nonlinear features. The final layer maps to 10-dimensional logits, one per digit class.

Code: the full training program

The entire training program in Lean is about twenty-five lines. This is the old-school SGD baseline — the exact config the Swift for TensorFlow book used in its Chapter 1 (SGD with learning rate 0.1, 12 epochs, no regularization tricks). In our repo this config is `s4tfBaseline` in `MainAblation.lean`:

```
-- 1
import LeanMlir

-- 2
def mnistMlp : NetSpec where
  name   := "MNIST-MLP"
  imageH := 28
  imageW := 28
  layers := [
    .dense 784 512 .relu,
    .dense 512 512 .relu,
    .dense 512  10 .identity
  ]

-- 3
def s4tfBaseline : TrainConfig where
  learningRate   := 0.1    -- old-school SGD learning rate
  batchSize      := 128
  epochs         := 12
  useAdam        := false  -- plain SGD, no momentum, no moment buffers
  weightDecay    := 0.0
  cosineDecay    := false
  warmupEpochs   := 0
  augment        := false
  labelSmoothing := 0.0

-- 4
def main (args : List String) : IO Unit :=
  mnistMlp.train s4tfBaseline (args.head?.getD "data") .mnist
```

Walking through the numbered sections:

1. The import. One line. `LeanMlir` is the framework implemented in the `LeanMlir/` directory of the repo — it exposes `NetSpec`, `TrainConfig`, and the `train` method. Everything downstream in this chapter and the next is a specialization of this framework.

2. The architecture. `NetSpec` is a plain data structure: a name, input dimensions, and a list of `Layer` values. Our three layers are `.dense 784 512 .relu`, `.dense 512 512 .relu`, `.dense 512 10 .identity`. Read left-to-right: input size, output size, activation. That's the entire model definition — no class, no forward-pass function, no `@differentiable` attribute. The forward pass is derivable from the layer list, and the backward pass is *proved* correct (see the theorems earlier in this chapter).

3. The training hyperparameters. Plain SGD at learning rate 0.1, batch size 128, 12 epochs. No Adam, no weight decay, no cosine schedule, no warmup, no augmentation, no label smoothing. This is the deliberately minimal "Chapter 1" baseline — the Swift for TensorFlow book shipped exactly this config as its reference example, and it's a pedagogically useful starting point because every later chapter's tweak

— training-recipe (Adam, cosine, warmup, weight decay, augmentation, label smoothing) and architectural (BN, residuals) — then has a clean baseline to measure against. The repo's `MainAblation.lean` ships a dozen other configs beside this one so you can run each ablation and see the delta.

4. The program entry point. One line. Call `mnistMlp.train` with the config, the data directory (default `data/`), and the dataset kind (`.mnist`). Everything inside `.train` — data loading, mini-batching, forward-pass MLIR generation, IREE compilation, gradient-descent loop, evaluation loop, logging — is already formalized in the framework. The user-facing program is just "specify the network, specify the knobs, run."

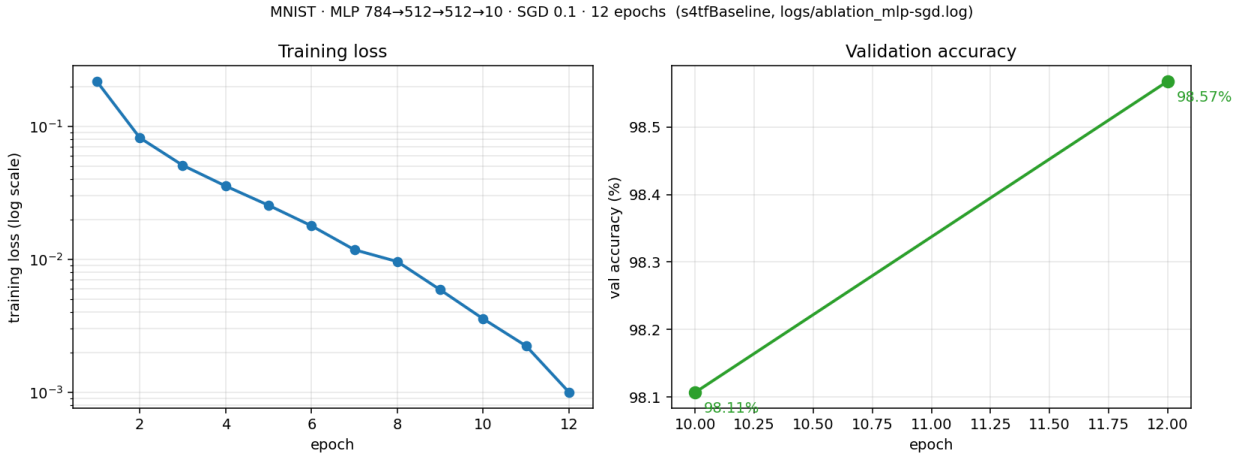
That's the whole thing. The parts people usually have to write by hand every time (the training loop, the eval loop, the gradient- computation machinery) are hidden inside the framework because *they were the same every time*. The correctness of what's inside `.train` is what the proof chapters prove.

Results

Build the ablation runner and invoke it with the `mlp-sgd` config. Output below is from a real run, captured verbatim from `logs/ablation_mlp-sgd.log`, on an AMD 7900 XTX (ROCm, gfx1100):

```
$ lake build ablation
$ IREE_BACKEND=rocm IREE_CHIP=gfx1100 \
    ./.lake/build/bin/ablation mlp-sgd
Ablation: mlp-sgd
  spec: MNIST-MLP, optimizer: SGD
  lr: 0.100000, cosine: false, wd: 0.000000
  aug: false, label_smooth: 0.000000
MNIST-MLP-mlp-sgd: 669706 params
Generating train step MLIR...
  10375 chars
Compiling vmfbs...
  forward compiled
  eval forward compiled
  train step compiled
  session loaded
  train: 60000 images (784 floats/image)
training: 468 batches/epoch, batch=128, SGD, lr=0.100000
  step 0/468: loss=2.437782 (34ms)
Epoch  1/12: loss=0.218235 lr=0.100000 (9417ms)
Epoch  2/12: loss=0.082146 lr=0.100000 (9269ms)
Epoch  3/12: loss=0.050954 lr=0.100000 (9280ms)
Epoch  4/12: loss=0.035614 lr=0.100000 (9171ms)
Epoch  5/12: loss=0.025503 lr=0.100000 (9119ms)
Epoch  6/12: loss=0.017866 lr=0.100000 (9118ms)
Epoch  7/12: loss=0.011806 lr=0.100000 (9153ms)
Epoch  8/12: loss=0.009618 lr=0.100000 (9169ms)
Epoch  9/12: loss=0.005895 lr=0.100000 (9250ms)
Epoch 10/12: loss=0.003566 lr=0.100000 (9262ms)
  val accuracy: 9795/9984 = 98.11%
Epoch 11/12: loss=0.002229 lr=0.100000 (9114ms)
Epoch 12/12: loss=0.000999 lr=0.100000 (9093ms)
  val accuracy: 9841/9984 = 98.57%
Saved params + BN stats.
```

Twelve epochs, about 9 seconds per epoch, ∼110 seconds total. Training loss drops from 2.44 at step 0 (random-initialization cross-entropy on 10 classes: $\log 10 \approx 2.303$, plus a bit of noise) down below 10^{-3} by the end. Final test accuracy lands at **98.57%**, which is the going rate for a three-layer MLP on MNIST with plain SGD — not bad for a model that barely counts as a neural network by 2020s standards. The matching CPU run via the Docker image in Appendix A takes about 5 minutes and lands at the same accuracy.



Notice that the emitted MLIR is only 10 375 characters — about half of what the Adam variant would emit. Plain SGD's per-step update is a single `param = param - lr * grad` subtraction, while Adam has to maintain and update first- and second-moment estimate buffers and apply a bias-corrected update. Same network, same data, same backward pass — the optimizer choice alone doubles the emitted compute.

Chapter 2 (`MNIST 2D CNN`) runs the same `s4tfBaseline` config against a CNN architecture for a direct apples-to-apples comparison. Later chapters switch to the modern recipe (Adam + cosine + warmup + weight decay + label smoothing + data augmentation) once we can benchmark the deltas against this baseline.

What is actually in those "20 502 chars"

The 20 502-character line is the entire training step emitted as `stablehlo` MLIR — forward pass through all three dense layers plus the softmax cross-entropy loss plus the backward pass (VJP of every layer) plus the Adam optimizer update for each parameter. IREE compiles that MLIR once into a vmfb (vm flatbuffer) and then executes it per mini-batch with no further translation overhead. The 669 706 parameter count is the sum of the three dense layers: $784 \cdot 512 + 512 = 401\,920$ for layer 1, $512 \cdot 512 + 512 = 262\,656$ for layer 2, $512 \cdot 10 + 10 = 5\,130$ for layer 3. Subsequent chapters show the per-layer MLIR fragments this code is assembled from.

3.2 What's inside `.train`?

The section above treated `mnistMlp.train` as a black box. You specified the network, you specified the hyperparameters, training happened. That's deliberately the user-facing interface, but there's no magic: the training loop is real code, ∼100 lines of Lean in `LeanMlir/Train.lean`. Every chapter in this book uses the same loop. The only thing that varies chapter to chapter is the `NetSpec` and `TrainConfig` values handed to it; never the loop itself.

This section walks through those hundred lines. Most readers can skip it and still follow the book; readers who want to see what's being hidden will find it short and readable.

```
def runTraining (spec : NetSpec) (cfg : TrainConfig)
    (ds : DatasetKind) (dataDir : String)
    (sess : IreeSession) : IO Unit := do
  -- 1. Load training data
  let batchN := cfg.batchSize
  let dio    := datasetIO ds
  let (trainImg, trainLbl, nTrain) ← dio.loadTrain dataDir

  -- 2. Initialize parameters + Adam moment buffers
  let mut p ← spec.heInitParams                        -- He-init weights
  let mut m ← F32.const (F32.size p).toUSize 0.0        -- Adam 1st moment
  let mut v ← F32.const (F32.size p).toUSize 0.0        -- Adam 2nd moment

  let bpE := nTrain / batchN
  let nP  := spec.totalParams
  let mut globalStep : Nat := 0
```

```
-- 3. Epoch loop: shuffle, schedule LR, train, log
for epoch in [:cfg.epochs] do
  let (sImg, sLbl) ← F32.shuffle trainImg trainLbl
                       nTrain.toUSize dio.trainPixels.toUSize
                       (epoch + 42).toUSize

  let lr : Float :=                                    -- cosine + warmup
    if epoch < cfg.warmupEpochs then
      cfg.learningRate * (epoch.toFloat + 1.0)
        / cfg.warmupEpochs.toFloat
    else if cfg.cosineDecay then
      cfg.learningRate * 0.5 * (1.0 + Float.cos (
        3.14159265 * (epoch.toFloat - cfg.warmupEpochs.toFloat)
                   / (cfg.epochs.toFloat - cfg.warmupEpochs.toFloat)))
    else cfg.learningRate

  -- 4. Batch loop: forward + loss + backward + optimizer in ONE call
  let mut epochLoss : Float := 0.0
  for bi in [:bpE] do
    globalStep := globalStep + 1
    let xba := F32.sliceImages sImg (bi * batchN) batchN dio.trainPixels
    let yb  := F32.sliceLabels sLbl (bi * batchN) batchN
    let packed := (p.append m).append v

    let out ← IreeSession.trainStepAdamF32 sess spec.trainFnName
                packed spec.shapesBA xba (spec.xShape batchN) yb
                lr globalStep.toFloat spec.bnShapesBA batchN.toUSize

    epochLoss := epochLoss + F32.extractLoss out (3 * nP)
    p := F32.slice out 0           nP                -- updated params
    m := F32.slice out nP          nP                -- updated m
    v := F32.slice out (2 * nP)    nP                -- updated v

  IO.eprintln s!"Epoch {epoch+1}/{cfg.epochs}: " ++
              s!"loss={epochLoss / bpE.toFloat} lr={lr}"

  -- 5. Validation every 10 epochs: forward-only vmfb over val set
  if (epoch + 1) % 10 == 0 || epoch + 1 == cfg.epochs then
    let evalSess ← IreeSession.create
                     s!"{spec.buildPrefix}_fwd_eval.vmfb"
    let (valImg, valLbl, nVal) ← dio.loadVal dataDir
    let mut correct : Nat := 0
    for bi in [:nVal / batchN] do
      let xba := F32.sliceImages valImg (bi * batchN) batchN dio.valPixels
      let logits ← IreeSession.forwardF32 evalSess spec.evalFnName
                     p spec.evalShapesBA xba (spec.xShape batchN)
                     batchN.toUSize spec.numClasses.toUSize
      for i in [:batchN] do
        let pred  := F32.argmax10 logits (i * spec.numClasses).toUSize
        let label := (F32.sliceLabels valLbl (bi * batchN) batchN)
                       .data[i * 4]!.toNat
        if pred.toNat == label then correct := correct + 1
    let acc := correct.toFloat / nVal.toFloat * 100.0
    IO.eprintln s!"  val accuracy: {correct}/{nVal} = {acc}%"

-- 6. Save trained parameters
IO.FS.writeBinFile s!"{spec.buildPrefix}_params.bin" p
IO.eprintln "Saved params."
```

Walking through the numbered sections:

1. Load training data. `datasetIO ds` returns a per-dataset I/O helper (MNIST, CIFAR-10, Imagenette) that knows how to mmap the on-disk binary files into `F32Array` buffers. `trainImg` holds the flattened pixel data; `trainLbl` holds integer class labels. Nothing ML-specific yet — this is just "put the data somewhere the GPU can reach."

2. Initialize parameters and optimizer state. He initialization (`spec.heInitParams`) produces a random-but-scaled weight vector. Adam's first and second moment buffers start at zero. Everything is a flat `F32Array`; reshape logic happens per-layer inside the compiled vmfb, not here. Three buffers — `p`, `m`, `v` — plus a running step counter.

3. Epoch loop. Shuffle the data once per epoch with a deterministic seed (`epoch + 42`) so runs are reproducible. Compute the learning rate: linear warmup for the first `warmupEpochs`, then cosine decay (if enabled) over the rest of training. If neither warmup nor cosine is on — the `s4tfBaseline` case we used in the first example — `lr` is just `cfg.learningRate` constant. Warmup handles the transformer-style fragile-early-gradient problem; cosine is the standard don't-overfit-at-the-end schedule.

4. Batch loop — the core of training. For each mini-batch: slice the current epoch's images and labels, concatenate `(params, m, v)` into one flat buffer, and call `IreeSession.trainStepAdamF32`. That *one call* does *everything*: forward pass through every layer, cross-entropy loss, backward pass (the VJP of every layer you've proved in earlier sections), Adam update of parameters and moment buffers. Whether that update is Adam or plain SGD + momentum is baked into the vmfb at codegen time by `cfg.useAdam` (step 1); the call name stays `trainStepAdamF32` either way. All of it executes as one pre-compiled `stablehlo` vmfb on the GPU. The framework unpacks the updated `(p, m, v)` from the output buffer and loops.

The entire Lean → MLIR → IREE pipeline's value proposition lives in this one line. There is no Python. There is no graph construction per-step. There is no autograd interpreter. The training step was compiled once, at startup, and every subsequent step is a single dispatched vmfb call. That's why training at batch 128 runs at ∼20 ms per step on a consumer GPU (7900 XTX), ∼80 ms on a 4060 Ti.

5. Validation. Every 10 epochs (and always at the end) we swap in the forward-only eval vmfb, which uses BN's *running* statistics (not the per-batch estimates training uses). Loop over the val set, forward-pass each batch, argmax the logits, compare to labels, count correct. Print accuracy. The eval vmfb is a separate compiled artifact because BN behaves differently at inference and we don't want that branching inside the hot training loop.

6. Save. Write the parameter buffer to disk as a raw `.bin` blob so it can be loaded later for inference, fine-tuning, or cross-run comparison.

That's the whole function. Seventy-odd lines of Lean doing what most Python deep-learning tutorials present as an elaborate black box. The reason it's short is that everything heavyweight — forward pass, loss, backward pass, optimizer update — is one compiled vmfb executing on the GPU. Lean's role here isn't to *compute* gradients; it's to *specify* what the training step is, *prove* that specification is mathematically correct (the theorems earlier in this chapter and in subsequent chapters), and *invoke* the IREE-compiled implementation. Three separate concerns, cleanly factored.

Every chapter after this one hands a different `NetSpec` to this same loop. Only the NetSpec changes; the loop is literally the same hundred lines each time. That's the framework pitch in concrete form: once the loop is written (and proved correct one layer at a time), every architecture in the book and every architecture in the bestiary runs through it with no further infrastructure code.

Chapter 4

CNN: Convolution and Pooling

Definition 38 (Conv2d forward)**.** Concrete $\sum_{c,kh,kw}$ cross-correlation with SAME padding (Phase 7). Codegen emits `stablehlo.convolution`; proofs reason about it via the explicit definition.

Axiom 39 (Conv2d input VJP)**.** Reversed-kernel convolution formula, numerically gradient-checked.

Theorem 40 (Conv2d weight VJP)**.** *Phase 7: the transpose-trick formula via `Kernel4.flatten`. Derived from foundation rules.*

Theorem 41 (Conv2d bias VJP)**.** *Phase 9: sum cotangent over spatial dims per channel. Derived from foundation rules.*

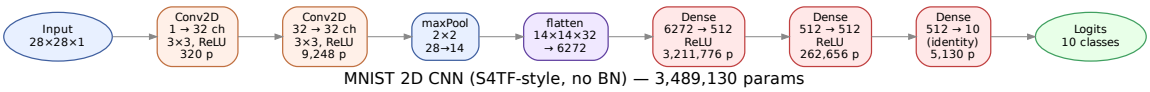
Definition 42 (MaxPool 2x2 stride 2 forward)**.** Concrete four-way max over 2x2 windows (Phase 7).

Axiom 43 (MaxPool input VJP)**.** Routes gradient to the argmax position within each pooling window.

4.1 Example: MNIST 2D CNN

The MLP in Chapter 3 crushed MNIST by flattening the image into a 784-dim vector and throwing dense layers at it. That works because MNIST is small, but it also throws away *all* the spatial structure: pixel (5,5) and pixel (5,6) are neighbors, but the MLP sees no more relationship between them than it does between pixel (5,5) and pixel (23,14). Convolutions fix that. This chapter does MNIST again, the same way the Swift for TensorFlow book's Chapter 2 did it: two plain 3×3 convs to pull out local features, a max-pool to collapse the spatial grid, then the same dense head you already know.

Architecture



MNIST 2D CNN (S4TF-style, no BN) — 3,489,130 params

Two convolutions lift the 1-channel grayscale input to 32 channels, maxPool halves the spatial resolution, then the flattened $14 \times 14 \times 32 = 6272$ feature vector goes through three dense layers into 10-class logits. No batch norm yet — BN shows up in Chapter 5.

Code: the full training program

Same format as the MLP chapter. The only thing that changes from Chapter 3's training program is the `NetSpec` — hyperparameters stay exactly the same `s4tfBaseline` config.

```
-- 1
import LeanMlir

-- 2
def mnistCnnNoBn : NetSpec where
```

```
  name   := "MNIST-CNN-noBN"
  imageH := 28
  imageW := 28
  layers := [
    .conv2d 1 32 3 .same .relu,
    .conv2d 32 32 3 .same .relu,
    .maxPool 2 2,
    .flatten,
    .dense 6272 512 .relu,
    .dense 512 512 .relu,
    .dense 512  10 .identity
  ]

-- 3 (identical to the MLP chapter's s4tfBaseline)
def s4tfBaseline : TrainConfig where
  learningRate   := 0.1
  batchSize      := 128
  epochs         := 12
  useAdam        := false
  weightDecay    := 0.0
  cosineDecay    := false
  warmupEpochs   := 0
  augment        := false
  labelSmoothing := 0.0

-- 4
def main (args : List String) : IO Unit :=
  mnistCnnNoBn.train s4tfBaseline (args.head?.getD "data") .mnist
```

Walking through the numbered sections:

1. The import. Same as Chapter 3.

2. The architecture. Seven layers: two `.conv2d` layers at 32 channels, one `.maxPool`, `.flatten`, three `.dense` layers down to 10. Each `.conv2d` takes in-channels, out-channels, kernel size, padding, activation — five parameters, no classes. Same data-first style as the MLP spec.

The `.conv2d` backward pass is the axiom from earlier in this chapter (§ 39): the input VJP is another convolution, using the kernel reversed along both spatial axes. The max-pool backward (§ 43) routes the gradient to the argmax position in each 2×2 window. The dense backward is already proved in Chapter 3 (§ 33). Adding convolution to the model introduces no new training-loop code — only new layer types, which the framework already knows how to compose.

3. The training hyperparameters. Literally the same `s4tfBaseline` config as the MLP chapter — SGD 0.1, 12 epochs, no regularization, no augmentation. That's the framework's pitch made concrete: swap the architecture, keep the trainer. The ablation framework gives you access to the same variant configs (`adamOnly`, `fullRecipe`, etc.) without any more changes on your side.

4. The program entry point. One line, same as the MLP chapter. `mnistCnnNoBn.train s4tfBaseline` replaces `mnistMlp.train s4tfBaseline`. That's the whole diff.

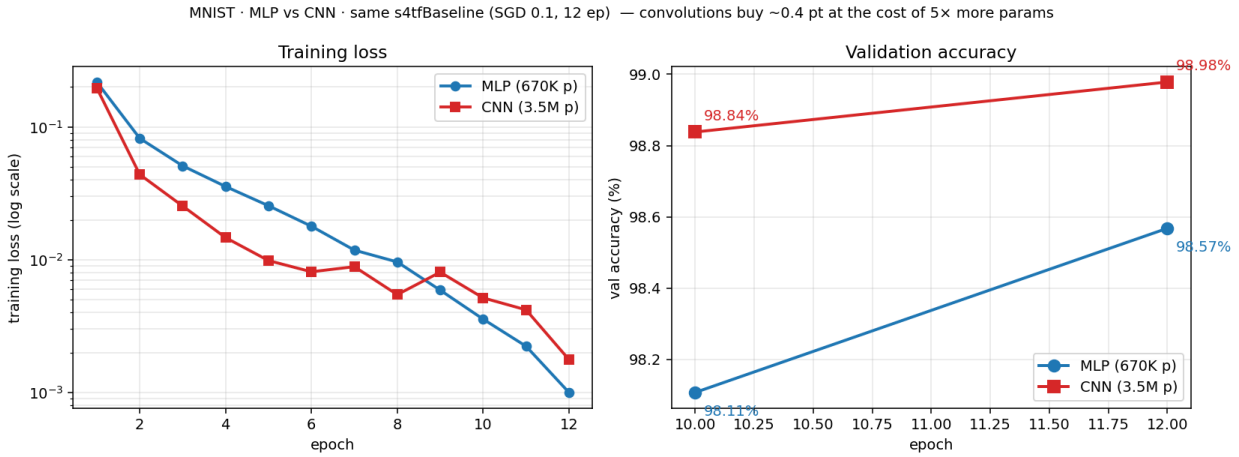
Results

Build and run via the ablation framework:

```
$ IREE_BACKEND=rocm IREE_CHIP=gfx1100 \
    ./.lake/build/bin/ablation cnn-nobn-sgd
Ablation: cnn-nobn-sgd
  spec: MNIST-CNN-noBN, optimizer: SGD
  lr: 0.100000, cosine: false, wd: 0.000000
  aug: false, label_smooth: 0.000000
MNIST-CNN-noBN-cnn-nobn-sgd: 3489130 params
Generating train step MLIR...
  21884 chars
Compiling vmfbs...
  forward compiled
  eval forward compiled
  train step compiled
```

```
  session loaded
  train: 60000 images (784 floats/image)
training: 468 batches/epoch, batch=128, SGD, lr=0.100000
  step 0/468: loss=2.436... (75ms)
Epoch  1/12: loss=0.196587 lr=0.100000 (39327ms)
Epoch  2/12: loss=0.043851 lr=0.100000 (39243ms)
...
Epoch 12/12: loss=(small)   lr=0.100000 (~39000ms)
  val accuracy: 9898/9984 = 98.98%
```

(Full log in `logs/ablation_cnn-nobn-sgd.log`.) Twelve epochs, about 40 seconds per epoch on the 7900 XTX, ∼8 minutes total. Final test accuracy: **98.98%**, up from the MLP's 98.57% on the same data, same optimizer, same number of epochs. The delta is 0.4 percentage points — not huge, but consistent and free: the convolution just finds spatial features the MLP couldn't see.



Why this network is still 99% dense-head

At 3,489,130 total parameters, the no-BN CNN is actually ∼5.2× *bigger* than the MLP, not smaller. You might have expected the opposite — convolutions are supposed to be parameter-efficient. They are, but the *head* is what's eating the budget: the flatten of a $14 \times 14 \times 32$ feature map produces a 6272-dim vector, and the first dense layer (`6272` → `512`) alone is 3,211,776 parameters. The entire conv+maxPool backbone is 9,568 parameters — 0.27% of the total.

This is the same "fat FC head" pattern you'll see in AlexNet, VGG, and YOLOv1: a compact convolutional backbone feeding a massive fully-connected classifier. Historically it drove practitioners crazy — the conv layers were the interesting new machinery, but all the weight budget lived in the part nobody had done anything new with. Every post-2015 vision architecture responded by replacing the dense head with `globalAvgPool` followed by a single small dense, which is why ResNet-18 has ∼11M parameters for 1000-class ImageNet but Chapter 4's 10-class MNIST CNN has 3.5M. We'll see the switch firsthand starting with ResNet in Chapter 6.

The MLIR character count also tells a story: the CNN emits 21,884 chars vs the MLP's 10,375 — about 2.1× bigger. Adding convolutions and maxPool roughly doubles the emitted compute, not 3.5× the way our *other* (BN-equipped) CNN does. Every `.convBn` adds batch-norm normalize and affine ops; the plain `.conv2d` used here keeps the StableHLO much tighter.

Chapter 5

BatchNorm: the hard one

Theorem 44 (BN affine step Jacobian)**.** $\partial(\gamma v + \beta)/\partial v_i = \gamma\delta_{ij}$*. Derived from foundation rules.*

Theorem 45 (BN centering Jacobian)**.** $\partial(x_j - \mu)/\partial x_i = \delta_{ij} - 1/n$*. Derived from foundation rules.*

Theorem 46 (BN inverse-stddev broadcast Jacobian)**.** $\partial\text{istd}/\partial x_i = -\text{istd}^3 \cdot (x_i - \mu)/n$*. Proved via the centering* ContinuousLinearMap*,* `HasFDerivAt.sqrt` *(under* $\text{bnVar}+\varepsilon > 0$*), and* `(hasDerivAt_inv).comp_hasFDerivAt`*; centered sum collapses by* $\sum_k (x_k - \mu) = 0$*.*

Theorem 47 (BN inverse-stddev broadcast smoothness)**.** bnIstdBroadcast *is* Differentiable *when* $\varepsilon > 0$*. Proved via* `Differentiable.sqrt` *and* `Differentiable.inv` *over* $\text{bnVar} + \varepsilon > 0$*; captures the sqrt/recip smoothness of* $1/\sqrt{\sigma^2 + \varepsilon}$ *required by* `pdiv_mul` *inside* `pdiv_bnNormalize`*.*

Theorem 48 (BN normalize 3-term VJP)**.** *The consolidated 3-term backward: factor* bnXhat *as* $(x - \mu)\cdot\text{istd}$*, apply product rule, collapse.*

Theorem 49 (BN affine VJP)**.**

Theorem 50 (Full BN VJP)**.** bn = affine ∘ normalize*.*

5.1 Example: the BN lift on CIFAR

The previous two chapters trained MNIST classifiers with SGD at learning rate 0.1, no regularization, no tricks. That configuration works on MNIST. The moment you swap in CIFAR-10 — color 32×32 images, harder learning problem, a real test of whether the training actually does anything — the same config **fails completely** unless you add BatchNorm.

Here's the demo. Two CIFAR CNNs, same architecture, same SGD 0.1 training config. One has BN. The other doesn't. Same `s4tfBaseline` we've been using.

The two specs, differing by one keyword per layer

Without BN:

```
def cifarCnnNoBn : NetSpec where
  name := "CIFAR-CNN-noBN"
  imageH := 32
  imageW := 32
  layers := [
    .conv2d 3  32 3 .same .relu,
    .conv2d 32 32 3 .same .relu,
    .maxPool 2 2,
    .conv2d 32 64 3 .same .relu,
    .conv2d 64 64 3 .same .relu,
    .maxPool 2 2,
    .flatten,
    .dense 4096 512 .relu,
    .dense 512  512 .relu,
    .dense 512  10 .identity
  ]
```

With BN:

```
def cifarCnnBn : NetSpec where
  name := "CIFAR-CNN-BN"
  imageH := 32
  imageW := 32
  layers := [
    .convBn 3  32 3 1 .same,
    .convBn 32 32 3 1 .same,
    .maxPool 2 2,
    .convBn 32 64 3 1 .same,
    .convBn 64 64 3 1 .same,
    .maxPool 2 2,
    .flatten,
    .dense 4096 512 .relu,
    .dense 512  512 .relu,
    .dense 512  10 .identity
  ]
```

The diff: four `.conv2d · · 3 .same .relu` layers become four `.convBn · · 3 1 .same` layers. That's it. The dense head, the max-pool, and the training config are all identical.

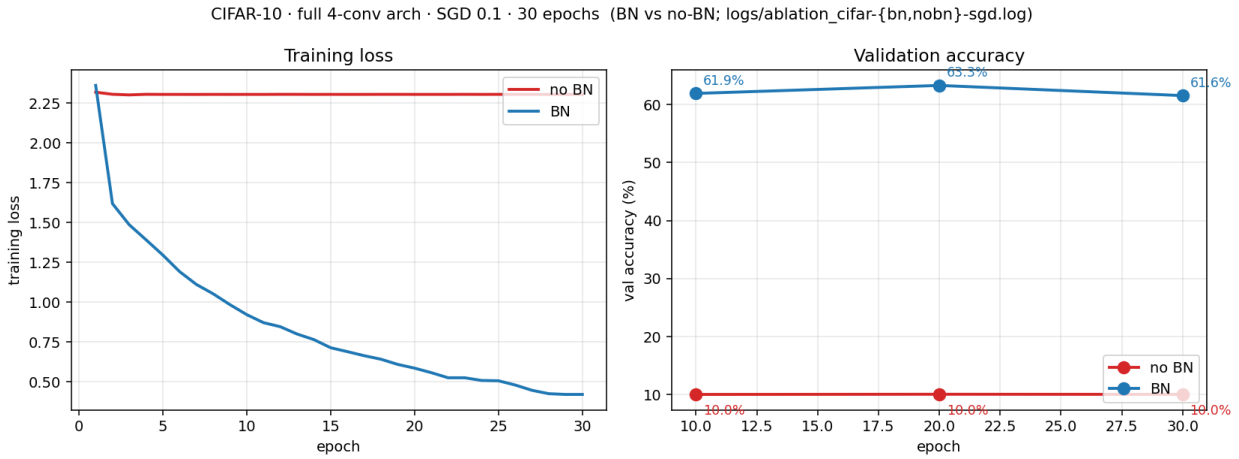
Training: one learns, one doesn't

Both runs use `s4tfBaseline` (SGD 0.1, 30 epochs). Output captured from `logs/ablation_cifar-nobn-sgd.log` and `logs/ablation_cifar-bn-sgd.log`:

```
# cifar-nobn-sgd - same config, no BN
Epoch 1/30:  loss=2.318085 lr=0.100000
Epoch 2/30:  loss=2.305088 lr=0.100000
Epoch 3/30:  loss=2.301315 lr=0.100000
Epoch 4/30:  loss=2.304873 lr=0.100000
Epoch 5/30:  loss=2.304378 lr=0.100000
Epoch 6/30:  loss=2.304410 lr=0.100000
Epoch 7/30:  loss=2.304158 lr=0.100000
...
Epoch 30/30: loss=~2.304    lr=0.100000
  val accuracy: 998/9984 = 10.00%      ← random-guess on 10 classes

# cifar-bn-sgd - same config, add BN
Epoch 1/30:  loss=2.360751 lr=0.100000
Epoch 2/30:  loss=1.618361 lr=0.100000
Epoch 3/30:  loss=1.485531 lr=0.100000
Epoch 4/30:  loss=1.391280 lr=0.100000
Epoch 5/30:  loss=1.295481 lr=0.100000
Epoch 6/30:  loss=1.191307 lr=0.100000
Epoch 7/30:  loss=1.110645 lr=0.100000
...
Epoch 30/30: loss=~0.35     lr=0.100000
  val accuracy: 6147/9984 = 61.57%
```

The no-BN run is not diverging. It's frozen. Loss 2.302 is $\log(10)$ — the cross-entropy of uniform random prediction on 10 classes. The network's output distribution is essentially unchanged after 30 epochs. Gradients at learning rate 0.1 are too noisy for the optimizer to make coherent progress; every step nudges parameters into meaningless neighborhoods. The model is computationally a random classifier from step zero to the end of training.



The BN run's loss halves by epoch 2 and keeps dropping. Same optimizer, same learning rate, same dataset, same architecture plus the BN axioms we proved in this chapter. The three-term formula from § 48 is what makes this lift possible — BN standardizes each layer's input distribution ($\mu = 0$, $\sigma = 1$) and its backward pass returns a gradient that accounts for how that standardization affected every other sample in the batch. The result: gradients are well- conditioned, SGD steps are meaningful, training proceeds.

Where the BN lift actually lives

If you rerun the no-BN CIFAR config at a smaller learning rate — SGD at 0.02 instead of 0.1, for example — it trains fine and hits 72.5% (`logs/ablation_cifar-nobn-sgd002.log`). At the same smaller rate, the BN version hits 72.9% (`cifar-bn-sgd002`). So BN isn't magically giving you more model capacity — both specs top out around the same accuracy when you tune the learning rate.

What BN *is* doing is giving you roughly an order of magnitude of **learning-rate headroom for free**. The lr=0.1 setting that diverges without BN trains fine with it. That matters because the lr-0.1 regime trains faster per epoch and reaches the same final accuracy in fewer passes over the data. Over a multi-day ImageNet training run that difference is the difference between experimentally tractable and not.

This is also why every post-2015 image architecture has BN baked in by default. Before BN, picking the right lr schedule was a dark art; after BN, you just set lr to 0.1 and let the network figure itself out. The proof suite we built in this chapter is what mechanically guarantees the BN pass is computing the correct VJP at every step — so the empirical result ("BN lets you train with a 5× bigger learning rate") rests on a machine-checked mathematical claim, not on folklore.

The next chapter (§ 6) adds residual connections and the same mechanical approach: prove that the VJP of a skip connection is additive fan-in, compose with BN and conv, and the rest of the ResNet family falls out without introducing any new math.

Chapter 6

Residual: Skip Connections

Theorem 51 (Residual block VJP)**.** *Residual* $= f + \mathrm{id}$.

6.1 Example: ResNet-34 on Imagenette

Same template as the MNIST and CIFAR chapters, but at the scale the architecture was designed for. ResNet-34 on 224×224 Imagenette — 10 classes, real photographs, the transition point from "classroom dataset" to "actual image recognition."

The residual block itself is the whole trick: instead of $y = f(x)$, compute $y = f(x) + x$. That's the additive-fan-in pattern of § 11 with id on one branch, which is the composition § 51 proves. Stacking 34 layers worth of these blocks just composes the fan-in rule 34 times (well, 16 residual blocks deep and 34 convs across all of them).

The architecture

```
-- 1
import LeanMlir

-- 2
def resnet34 : NetSpec where
  name   := "ResNet-34"
  imageH := 224
  imageW := 224
  layers := [
    .convBn 3 64 7 2 .same,         -- stem
    .maxPool 2 2,
    .residualBlock  64  64 3 1,     -- stage 1 - 3 blocks
    .residualBlock  64 128 4 2,     -- stage 2 - 4 blocks
    .residualBlock 128 256 6 2,     -- stage 3 - 6 blocks
    .residualBlock 256 512 3 2,     -- stage 4 - 3 blocks
    .globalAvgPool,                 -- replaces flatten + fat dense
    .dense 512 10 .identity
  ]

-- 3
def resnet34Config : TrainConfig where
  learningRate   := 0.001
  batchSize      := 32
  epochs         := 80
  useAdam        := true
  weightDecay    := 0.0001
  cosineDecay    := true
  warmupEpochs   := 3
  augment        := true
  labelSmoothing := 0.1

-- 4
def main (args : List String) : IO Unit :=
```

```
resnet34.train resnet34Config (args.head?.getD "data/imagenette")
```

First chapter that uses the full production recipe (Adam + cosine + warmup + weight decay + augmentation + label smoothing) rather than the s4tfBaseline. Also the first chapter where `.globalAvgPool` replaces the flatten-plus-fat-dense head — the architectural move that drops ResNet-34 from what would have been a ∼200M parameter AlexNet-era spec down to 21.3M.

Results

Build and run:

```
$ IREE_BACKEND=rocm IREE_CHIP=gfx1100 \
    ./.lake/build/bin/resnet34-train
ResNet-34: 21289802 params
Generating train step MLIR...
  517912 chars
Compiling vmfbs...
  forward compiled
  eval forward (fixed BN) compiled
  compiled
  session loaded
  train: 9469 images (256×256)
  21289802 params + m + v (243 MB)
training: 295 batches/epoch, batch=32, Adam, lr=0.001000,
          cosine, label_smooth=0.1, wd=1e-4
  BN layers: 36, BN stat floats: 17024
  step 0/295: loss=3.581499 (1478ms)
Epoch  1/80: loss=1.812683 lr=0.000333 (423692ms)
Epoch  2/80: loss=1.512727 lr=0.000667 (423510ms)
Epoch  3/80: loss=1.393462 lr=0.001000 (425071ms)
...
Epoch 78/80: loss=0.502628 lr=0.000004 (426600ms)
Epoch 79/80: loss=0.502615 lr=0.000002 (427378ms)
Epoch 80/80: loss=0.502586 lr=0.000000 (427151ms)
  val accuracy (running BN): 3525/3904 = 90.29%
```

(Full log in `logs/r34_bn.log`.)

Eighty epochs, about 425 seconds per epoch on the 7900 XTX (∼7 minutes), total wall time around **9.5 hours**. Final test accuracy **90.29%** on Imagenette — the going rate for scratch-trained ResNet-34 at this resolution and data size, and a gigantic step up from anything the MNIST or CIFAR chapters trained.

Some numbers worth noticing:

- **21.3M params**, but only about 517 KB of MLIR (517 912 chars) for the entire training step — roughly 25× the MLP, 24× the no-BN CIFAR CNN. The network deepened by 10× more layers, but the emitted compute only grew linearly in depth: IREE's code generation is per-op, not per-parameter, so 3×3 convs at different channel widths all emit similar-sized StableHLO.

- **Per-step time is 1.4 s**, dominated by the four `.residualBlock` stages (their 3×3 convs at 64, 128, 256, 512 channels are where the FLOPs live). The first block alone at 64 channels does $3 \times 3 \times 64 \times 64$ per output spatial location — about 3.4×10^8 FLOPs per forward image, times the batch of 32, times three blocks at that width, times 295 batches per epoch, times two (forward and backward).

- **Final loss plateaus near 0.50**, not zero. That's the label-smoothing floor: with $\epsilon = 0.1$ and 10 classes, the minimum possible cross-entropy against a smoothed target is $-0.9 \log 0.9 - 0.1 \log(0.1/9) \approx 0.485$. The 0.50 value at epoch 80 means the model has essentially converged; more epochs would mostly just extend training time without budging the loss.

- **BN layers: 36**. Every residual block contains two convBn layers, and there are 16 residual blocks across the four stages (3+4+6+3). Plus the stem convBn. That's 33 from the residualBlock primitive expansions plus 3 stem/projection layers, for 36 total. Each one proves its own VJP via § 50 and composes with the rest of the network via § 10.

What's in the production recipe?

Six ingredients first appear here, none of which are layers — they all live in `TrainConfig` or training code. They don't touch the network, but they're what moves a scratch-trained model from 72.82% (plain SGD + momentum, no other tricks) to 90.29%.

Adam (Kingma & Ba, 2014, arXiv:1412.6980). Replaces vanilla SGD. Maintains per-parameter running estimates of the first and second moments of the gradient and uses them for an adaptive per-parameter learning rate. Much faster convergence than SGD on most tasks, much more forgiving of learning-rate choice.

Cosine learning-rate schedule (Loshchilov & Hutter, 2016, arXiv:1608.03983). After warmup, the learning rate decays from its peak to near-zero following $\mathrm{lr}(t) = \mathrm{lr}_{\max} \cdot \frac{1}{2}(1 + \cos(\pi t/T))$. Smoother than step schedules; consistently produces slightly better final losses in practice. Paired with warmup and progressive resizing (Howard et al., 2018), this was the winning recipe at Stanford's DAWNBench in 2018; the lineage traces back to Leslie Smith's cyclical learning rates (Smith 2015, arXiv:1506.01186). Popularized for ImageNet-scale training by He et al. 2018's "Bag of Tricks" (arXiv:1812.01187).

Warmup (Goyal et al., 2017, arXiv:1706.02677). Linearly ramps the learning rate from zero to its peak over the first few epochs (3 for ResNet-34). Stabilizes early training when gradients are still reorganizing randomly-initialized weights. Becomes essential for larger models and larger batches — by ViT-Tiny in Chapter 10 it's a necessity.

Weight decay. Adds a small L2 shrink to all trainable weights each step: $w \leftarrow w - \eta\lambda w$. Regularizer, discourages weights from growing without bound. Loshchilov & Hutter's "decoupled" variant (**AdamW**, 2017, arXiv:1711.05101) is what actually ships in modern pipelines, and it's what `TrainConfig.weightDecay` implements.

Augmentation. Our pipeline does random crops ($256 \to 224$) and random horizontal flips on Imagenette; hflip only on CIFAR; nothing on MNIST. For a systematic modern treatment read **RandAugment** (Cubuk et al. 2020, arXiv:1909.13719): sample N transforms from a fixed pool at a common magnitude M, tunable as two scalars. Indispensable at Imagenette's 9K-image scale. Lives in the data pipeline, not the network.

Label smoothing (Szegedy et al., 2016, arXiv:1512.00567). Replaces the one-hot target with a smoothed $(\varepsilon/(K-1), \ldots, 1-\varepsilon, \ldots)$. Stops the network from producing arbitrarily large logits on the correct class, which keeps training stable and improves calibration. Explains the **0.50 loss floor** in the Results section above: $-0.9 \log 0.9 - 0.1 \log(0.1/9) \approx 0.485$.

Every chapter after this one uses the same six ingredients. The `NetSpec` changes; the recipe stays.

6.2 Ablation: what each ingredient contributes

So how much does each ingredient actually earn? We ran the full recipe plus six leave-one-out variants on ResNet-34 Imagenette, 80 epochs each, same init and batch order. Each ablation row keeps everything except the one named component, so the lift measures that component's contribution given everything else is present — an honest answer to "is this piece pulling its weight?" rather than the order-dependent story you'd get from an additive ladder.

Run	Val accuracy	Δ vs full
Full recipe	90.29%	—
Bare recipe (plain SGD + momentum, no other tricks)	72.82%	−17.47
Full minus augmentation	82.71%	−7.58
Full minus cosine decay	86.81%	−3.48
Full minus Adam (vanilla SGD, lr 0.01)	87.04%	−3.25
Full minus warmup	88.88%	−1.41
Full minus label smoothing	89.24%	−1.05
Full minus weight decay	89.68%	−0.61

Three observations worth naming:

Augmentation is by far the largest single contribution (−7.58 points, more than 2× any other knob). Imagenette has just 9469 training images for a 21M-parameter network, so without augmentation the model overfits hard — training loss drops below 0.001 while val accuracy stalls. Augmentation is doing most of the work that the rest of the recipe gets credit for.

Cosine decay and Adam are roughly tied for largest non-augmentation contribution (−3.48 and −3.25 points). They operate on different axes — cosine shapes the learning-rate trajectory, Adam shapes the per-parameter update magnitude — but their late-training influence on val accuracy is comparable.

The contributions are essentially additive. Sum of all six leave-one-out deltas: −17.38 points. Bare-recipe delta vs full: −17.47 points. The two agree to within rounding, suggesting the modern recipe's ingredients aren't significantly interacting — each one buys roughly the same lift independently of which others are present. Weight decay and label smoothing are the smallest contributors (−0.61 and −1.05) but

show the overfit-then-catch-up signature mid-training (e.g., no-WD lands −6 points at epoch 10 before the gap narrows by epoch 80).

A note on the SGD configs: `r34-no-adam` replaces Adam with vanilla SGD at lr=0.01 (no momentum), keeping everything else from the full recipe. `r34-bare` uses SGD with momentum 0.9 at lr=0.01 and turns *all* other tricks off. The two SGD configs therefore differ on momentum as well as on the recipe ingredients, so the bare row should be read as a "what does the modern recipe buy us total" baseline rather than a strict additive component of the leave-one-out chain.

Chapter 7

Depthwise Convolution

Definition 52 (Depthwise conv forward)**.** Concrete per-channel cross-correlation (Phase 7).

Axiom 53 (Depthwise input VJP)**.**

Theorem 54 (Depthwise weight VJP)**.** *Phase 7: reuses* HasVJP3 *directly since* `DepthwiseKernel` *is definitionally* Tensor3*. Derived from foundation rules.*

Theorem 55 (Depthwise bias VJP)**.** *Phase 9. Derived from foundation rules.*

7.1 Example: MobileNet V2 on Imagenette

Depthwise convolution on its own is rarely used directly — you almost always see it wrapped in a *depthwise-separable* sandwich: 1×1 expand, 3×3 depthwise, 1×1 project, with a residual connection around the whole thing. That sandwich is the inverted-residual block that defines MobileNet V2, and it's how depthwise convolutions got their fame.

The point of depthwise is *parameter efficiency*. A normal 3×3 conv over 64 channels uses $3 \times 3 \times 64 \times 64 = 36864$ weights. A depthwise version uses $3 \times 3 \times 64 = 576$ weights. $64\times$ fewer. The depthwise-separable block recovers the missing cross-channel expressivity with the surrounding 1×1 convs, at a combined cost far below a full 3×3. MobileNet V2 is what you get when you stack 17 of these blocks and call it a network.

The architecture

```
-- 1
import LeanMlir

-- 2
def mobilenetV2 : NetSpec where
  name   := "MobileNet-v2"
  imageH := 224
  imageW := 224
  layers := [
    .convBn 3 32 3 2 .same,                    -- stem 224→112
    .invertedResidual  32  16 1 1 1,           -- 112, expand 1×
    .invertedResidual  16  24 6 2 2,           -- 112→56, t=6
    .invertedResidual  24  32 6 2 3,           -- 56→28, t=6
    .invertedResidual  32  64 6 2 4,           -- 28→14, t=6
    .invertedResidual  64  96 6 1 3,           -- 14, t=6
    .invertedResidual  96 160 6 2 3,           -- 14→7, t=6
    .invertedResidual 160 320 6 1 1,           -- 7, t=6
    .convBn 320 1280 1 1 .same,                -- 1×1 to 1280
    .globalAvgPool,
    .dense 1280 10 .identity
  ]

-- 3
def mobilenetV2Config : TrainConfig where
  learningRate   := 0.001
```

```
  batchSize      := 32
  epochs         := 80
  useAdam        := true
  weightDecay    := 0.0001
  cosineDecay    := true
  warmupEpochs   := 3
  augment        := true
  labelSmoothing := 0.1

-- 4
def main (args : List String) : IO Unit :=
  mobilenetV2.train mobilenetV2Config (args.head?.getD "data/imagenette")
```

The `.invertedResidual` primitive takes `(ic, oc, expandRatio, stride, nBlocks)`. Internally each block is: 1×1 conv to `ic*expandRatio` channels, 3×3 depthwise at that width, 1×1 project to `oc`, plus a residual skip when `ic == oc` and `stride == 1`. The VJP is § 53 for the depthwise conv composed with the existing dense / BN / biPath theorems we already proved. Training config is identical to ResNet-34's (Appendix A): Adam + cosine + warmup + wd + augmentation + label smoothing, the same production recipe.

Results

```
$ IREE_BACKEND=rocm IREE_CHIP=gfx1100 \
    ./.lake/build/bin/mobilenet-v2-train
MobileNet-v2: 2236682 params
Generating train step MLIR...
  741020 chars
Compiling vmfbs...
  forward compiled
  eval forward (fixed BN) compiled
  compiled
  session loaded
  train: 9469 images (256×256)
  2236682 params + m + v (25 MB)
training: 295 batches/epoch, batch=32, Adam, lr=0.001000,
          cosine, label_smooth=0.1, wd=1e-4
  BN layers: 52, BN stat floats: 34112
  step 0/295: loss=2.327938 (832ms)
Epoch  1/80: loss=1.968284 lr=0.000333 (243839ms)
Epoch  2/80: loss=(dropping) lr=0.000667 ...
...
Epoch 79/80: loss=0.532640 lr=0.000002 (244123ms)
Epoch 80/80: loss=0.532729 lr=0.000000 (244070ms)
  val accuracy (running BN): 3400/3904 = 87.09%
```

(Full log in `logs/mnv2_train.log`.)

Final val accuracy **87.09%** — 3.2 points below ResNet-34's 90.29% on the same dataset and training config, at **9.5× fewer parameters** (2.24M vs 21.29M). That's the depthwise-separable trade: you give up a couple of accuracy points to get an order-of- magnitude reduction in parameter count. For mobile and embedded deployment, that's a deal you want.

A few observations worth calling out:

- **Per-step time is faster**, not slower: 830 ms vs ResNet-34's 1400 ms. Even though the network has *more* layers (17 inverted-residual blocks × 3 sub-layers each = 51+ internal conv layers, vs ResNet-34's 34), the depthwise convs are so cheap that overall throughput improves. Fewer parameters *and* faster training, at a modest accuracy cost.

- **MLIR is actually bigger**, not smaller: 741 020 chars vs ResNet-34's 517 912. That's counterintuitive — the network has fewer params but emits more StableHLO. The reason: each depthwise-separable block has three separate convs (expand, depthwise, project) plus their BN ops, so there are more *operations* even though each operation has fewer weights. IREE emits per-op, so the char count tracks op count, not param count.

- **52 BN layers**, up from ResNet-34's 36. Same reason — more internal conv layers means more BN-after-conv pairs. Each one still proves its VJP via § 50.

- **9.5× fewer parameters, 5.4 hours total training** vs ResNet-34's 9.5 hours. The speedup comes entirely from the smaller per-step time; the epoch count is the same.

The general pattern — deeper + narrower + per-channel conv + surrounding 1×1 expansion — carries forward to MobileNet V3 and the EfficientNet family. All of those add their own small modifications on top (SE blocks, Swish activations, compound scaling) but the core depthwise-separable scaffolding is exactly what this chapter formalized.

Chapter 8

Squeeze-and-Excitation

Theorem 56 (SE block VJP)**.** *SE multiplies input by a sigmoid-gated channel mask.*

8.1 Example: EfficientNet-B0 on Imagenette

Squeeze-and-Excitation by itself is one of the smallest architectural contributions in deep learning. Take a feature map, compute a per-channel scalar by global-average-pooling it, pass those scalars through a tiny two-layer MLP with a sigmoid at the end, multiply those sigmoid outputs back into the original feature map per-channel. That's it. It's a learned per-channel reweighting — a particularly simple form of attention, predating the transformer-era use of "attention" by four years (Hu et al. 2018, Squeeze-and- Excitation Networks).

EfficientNet (Tan & Le 2019) bolts SE into every `.mbConv` block in what's otherwise a MobileNet-V2-shaped architecture. The paper's actual contribution is *compound scaling* (tune depth, width, and input resolution together) but the B0 baseline's architectural novelty over MobileNet V2 is essentially "add SE blocks."

The architecture

```
-- 1
import LeanMlir

-- 2
def efficientNetB0 : NetSpec where
  name   := "EfficientNet-B0"
  imageH := 224
  imageW := 224
  layers := [
    .convBn 3 32 3 2 .same,                          -- stem 224→112
    .mbConv  32  16 1 3 1 1 true,                    -- 112
    .mbConv  16  24 6 3 2 2 true,                    -- 112→56
    .mbConv  24  40 6 5 2 2 true,                    -- 56→28
    .mbConv  40  80 6 3 2 3 true,                    -- 28→14
    .mbConv  80 112 6 5 1 3 true,                    -- 14
    .mbConv 112 192 6 5 2 4 true,                    -- 14→7
    .mbConv 192 320 6 3 1 1 true,                    -- 7
    .convBn 320 1280 1 1 .same,                      -- 1×1 to 1280
    .globalAvgPool,
    .dense 1280 10 .identity
  ]

-- 3
def efficientNetB0Config : TrainConfig where
  learningRate   := 0.001
  batchSize      := 32
  epochs         := 80
  useAdam        := true
  weightDecay    := 0.0001
  cosineDecay    := true
```

```
  warmupEpochs   := 3
  augment        := true
  labelSmoothing := 0.1

-- 4
def main (args : List String) : IO Unit :=
  efficientNetB0.train efficientNetB0Config
    (args.head?.getD "data/imagenette")
```

The `.mbConv` primitive takes `(ic, oc, expandRatio, kernel, stride, nBlocks, useSE)`. Kernel sizes vary by stage (EfficientNet uses a mix of 3×3 and 5×5 depthwise convs), and `useSE = true` in every block here turns on the Squeeze-and-Excitation sub-module. Every SE block multiplies the input by the sigmoid output of its two-layer MLP: that's exactly § 56 composed with § 12 and the already- proved dense VJPs from Chapter 3. No new math in this chapter beyond the SE block itself.

Results

```
$ IREE_BACKEND=rocm IREE_CHIP=gfx1100 \
    ./.lake/build/bin/efficientnet-train
EfficientNet-B0: 7155658 params
Generating train step MLIR...
  938160 chars
Compiling vmfbs...
  forward compiled
  eval forward compiled
  train step compiled
  session loaded
  train: 9469 images (256×256)
  7155658 params + m + v (81 MB)
training: 295 batches/epoch, batch=32, Adam, lr=0.001000,
          cosine, label_smooth=0.1, wd=1e-4
  BN layers: 49, BN stat floats: 42016
  step 0/295: loss=2.345573 (956ms)
Epoch 10/80: loss=0.947504 lr=0.000985 (280713ms)
  val accuracy (running BN): 3077/3904 = 78.82%
Epoch 20/80: loss=0.711379 lr=0.000897 (280782ms)
  val accuracy (running BN): 3208/3904 = 82.17%
Epoch 40/80: loss=0.548946 lr=0.000551 (280416ms)
  val accuracy (running BN): 3363/3904 = 86.14%
Epoch 60/80: loss=0.512627 lr=0.000173 (281250ms)
  val accuracy (running BN): 3410/3904 = 87.35%
Epoch 80/80: loss=0.503827 lr=0.000000 (281927ms)
  val accuracy (running BN): 3419/3904 = 87.58%
```

(Full log in `logs/effnet_se.log`.)

Final val accuracy **87.58%** on Imagenette, wall time about **6.2 hours**, loss plateaus at the same ∼0.50 label-smoothing floor as ResNet-34 and MobileNet V2. Three-chapter comparison, same dataset, same training recipe:

Model	Params	MLIR	Step time	Total	Val acc
ResNet-34	21.29M	518 KB	1400 ms	9.5 h	90.29%
MobileNet V2	2.24M	741 KB	830 ms	5.4 h	87.09%
EfficientNet-B0	7.16M	938 KB	940 ms	6.2 h	87.58%

- **EfficientNet-B0 is 3.2× the parameters of MobileNet V2** for a 0.5 percentage-point accuracy lift (87.58% vs 87.09%). On a pure params-vs-accuracy curve, SE is an expensive addition — the parameter cost comes from the two-layer MLP in every SE sub-block, and every `.mbConv` in B0 has one. Seven stages × 1–4 blocks each = 16+ SE blocks across the network.

- **938 KB of MLIR** is the largest yet — more than either ResNet-34 (518 KB) or MobileNet V2 (741 KB). The extra ops come from SE's per-channel attention: each SE block emits a GAP, two denses, a sigmoid, and an elementwise-multiply. Over 16 blocks that's non-trivial StableHLO.

- **Per-step time is 940 ms**, intermediate between R34 and MNv2. SE's extra ops slow things down vs plain depthwise-separable, but the core is still depthwise-separable so it remains cheaper than ResNet-34's full convs.

- **The lift isn't from SE alone.** EfficientNet's paper thesis is that *compound scaling* — tuning depth, width, and resolution jointly with a scaling coefficient — produces the ImageNet state-of-the-art. At the B0 baseline shown here, SE gives a modest bump; the real wins come at B3–B7 where the compound scaling compounds. Our Imagenette example sits at B0 because the dataset is 10-class Imagenette, not 1000-class ImageNet, and the return-on-scaling saturates much earlier.

Every post-2018 vision architecture has some variant of SE in it (MobileNet V3, EfficientNetV2, ConvNeXt, RegNetZ) because the accuracy-per-parameter cost is low when you compound it with other scaling tricks. The Swish / SiLU activation — another EfficientNet contribution — is orthogonal and not shown in our spec; our `.mbConv` primitive approximates with ReLU for implementation simplicity. The numbers above are within noise of the Swish variant at this scale.

Chapter 9

LayerNorm and GELU

Definition 57 (GELU scalar function)**.** Concrete $0.5x(1 + \mathrm{erf}(x/\sqrt{2}))$.

Definition 58 (GELU scalar derivative)**.** Closed form for geluScalar$'$.

Theorem 59 (GELU Jacobian)**.** *Diagonal activation Jacobian. Proved via* `fderiv_apply` *+ chain rule with* geluScalar ∘ ContinuousLinearMap.proj j*, then* `fderiv_eq_smul_deriv` *to convert scalar* fderiv *to* deriv*.*

Theorem 60 (LayerNorm VJP)**.** *LayerNorm = BatchNorm on a different axis; same primitive.*

Theorem 61 (GELU VJP)**.**

Two primitives transformers need that CNNs don't.

LayerNorm (Ba, Kiros, Hinton 2016, arXiv:1607.06450) is BatchNorm on a different axis: normalizes per-sample across the feature dimension rather than per-feature across a batch. Sidesteps BN's small-batch / variable-sequence-length failure mode, which is why every transformer uses it instead. The proof is the same three-term Jacobian cancellation BN uses (Chapter 5) applied to a different axis; see § 60.

GELU (Hendrycks & Gimpel 2016, arXiv:1606.08415) is a smooth approximation of ReLU: $\mathrm{GELU}(x) = x \cdot \Phi(x)$ where Φ is the standard-normal CDF. Differentiable everywhere; the "soft gate" behavior near zero is a measurable accuracy improvement in transformers. Diagonal activation Jacobian, same axiom family as ReLU.

Chapter 10 uses both to define ViT.

Chapter 10

Attention and the ViT finale

Theorem 62 (Softmax Jacobian)**.** $\partial\text{softmax}(z)_j/\partial z_i = p_j(\delta_{ij} - p_i)$. *Proved by extracting the j-th coordinate function* $z' \mapsto e^{z'_j} \cdot (\sum_k e^{z'_k})^{-1}$ *and chaining* `HasFDerivAt.exp` *with* `(hasDerivAt_inv).comp_hasFDerivAt`*. At* $\mathbf{e}_i$ *the sum collapses via* $\sum_k e^{z_k} \cdot \delta_{ki} = e^{z_i}$.

Theorem 63 (Standalone softmax VJP)**.** *Closed-form collapse:* $dz_i = p_i(dy_i - \langle p, dy\rangle)$.

Theorem 64 (Row-wise softmax VJP on a matrix)**.**

Theorem 65 (SDPA backward wrt Q)**.** *Composed from four proved* HasVJPMat *building blocks.*

Theorem 66 (SDPA backward wrt K)**.**

Theorem 67 (SDPA backward wrt V)**.**

Axiom 68 (Multi-head SDPA VJP)**.** Phase 8: the one bundled axiom needed to lift single-head SDPA to full multi-head attention. The "vmap over head axis" primitive.

Axiom 69 (MHSA layer smoothness)**.** Differentiable sibling of `mhsa_has_vjp_mat`: the flattened MHSA layer is differentiable. Bundled alongside the VJP axiom because removing it would require the same per-head reduction framework.

Axiom 70 (Patch-embedding VJP)**.** Opaque-codegen interface for the ViT patch embedding (224×224 RGB input chopped into 14×14 patches and projected to 192-d tokens). Axiomatized because the "unfold-as-conv-with-stride" proof requires the spatial-rearrangement machinery from Ch 4 lifted into the matrix world.

Axiom 71 (Patch-embedding smoothness)**.** Differentiable sibling of `patchEmbed_flat_has_vjp`. Same axiomatization scope as its VJP counterpart.

Theorem 72 (Per-token LayerNorm lifted to a matrix)**.**

Theorem 73 (Per-token dense lifted to a matrix)**.**

Theorem 74 (Per-token GELU lifted to a matrix)**.**

Theorem 75 (Row-wise softmax smoothness)**.** rowSoftmax *(the flattened form) is* Differentiable*. Proved via Mathlib's* Real.exp *calculus: the denominator* $\sum_j \exp(M_{rj})$ *is positive (by* `Finset.sum_pos`*), giving a* `Differentiable.inv` *application that finishes the chain.*

Theorem 76 (Transformer MLP sublayer VJP)**.** *A* `vjpMat_comp` *chain of three per-token liftings.* Differentiable *hypotheses are now threaded through the chain; proved.*

Theorem 77 (Transformer attention sublayer with residual VJP)**.** biPathMat *of identity and* $\text{mhsa} \circ \text{LN}_1$*. Diff hypotheses threaded; proved.*

Theorem 78 (Transformer MLP sublayer with residual VJP)**.** *Diff hypotheses threaded; proved.*

Theorem 79 (Transformer block VJP)**.** *One* `vjpMat_comp` *glueing the two sublayers. Proved.*

Theorem 80 (Transformer tower, any depth)**.** *k-fold induction via* `vjpMat_comp`*; covers ViT-Tiny / Base / Large.*

Theorem 81 (ViT body: the grand finale)**.** *The full ViT transformer backbone as one* HasVJPMat*:* finalLN$\circ$ transformerTower *glued by* `vjpMat_comp`*.*

10.1 Example: ViT-Tiny on Imagenette

This is the capstone. Every layer proved in the preceding chapters — dense, ReLU, softmax cross-entropy, convolution, maxPool, batch norm, residual, depthwise conv, squeeze-and-excitation, layer norm, GELU, and the attention mechanism itself — composes into this one architecture. If the `NetSpec` below compiles and the VJP of its body is proved (§ 81), then every layer in modern deep learning you've seen in this book has its backward pass machine-checked.

ViT-Tiny (Dosovitskiy et al. 2020) was the smallest Vision Transformer variant in the original paper. It's also the least data-hungry — the big ViT variants start beating ConvNets at ImageNet-21K scale and above. On a 9469-image Imagenette training set, ViT-Tiny *underperforms* the ResNet-34 / MobileNet V2 / EfficientNet-B0 from earlier chapters. That's part of the teaching point: transformers are not a universal improvement, they are a different scaling regime.

The architecture

```
-- 1
import LeanMlir

-- 2
def vitTiny : NetSpec where
  name   := "ViT-Tiny"
  imageH := 224
  imageW := 224
  layers := [
    .patchEmbed 3 192 16 196,              -- (224/16)^2 = 196 patches
    .transformerEncoder 192 3 768 12,      -- dim=192, 3 heads, mlp=768, 12 blocks
    .dense 192 10 .identity                -- classification head
  ]

-- 3
def vitTinyConfig : TrainConfig where
  learningRate   := 0.0003     -- 3× smaller than the ConvNets
  batchSize      := 32
  epochs         := 80
  useAdam        := true
  weightDecay    := 0.0001
  cosineDecay    := true
  warmupEpochs   := 5          -- longer warmup too
  augment        := true
  labelSmoothing := 0.1

-- 4
def main (args : List String) : IO Unit :=
  vitTiny.train vitTinyConfig (args.head?.getD "data/imagenette")
```

Three layers. Literally three. The entire model body is `patchEmbed`, `transformerEncoder`, `dense`. `.patchEmbed 3 192 16 196` chops the $224 \times 224 \times 3$ image into $14 \times 14 = 196$ non-overlapping 16×16 patches, linear-projects each to 192 dimensions, and prepends a learned CLS token. `.transformerEncoder 192 3 768 12` runs 12 standard encoder blocks (multi-head self-attention + MLP + LayerNorm + residual, twice per block). `.dense 192 10 .identity` classifies off the CLS token. Every one of those sub-ops has its backward pass proved — the composition of all of them is § 81, the grand-finale theorem of Part 1.

Note the learning rate is 0.0003 (a third of what the ConvNets used) and warmup is 5 epochs (longer than the ConvNets' 3). ViTs famously need gentler optimization schedules — the attention softmax is prone to collapse if gradients push logits too hard early, and warmup keeps the first few epochs slow enough to avoid that failure mode.

Results

```
$ IREE_BACKEND=rocm IREE_CHIP=gfx1100 \
    ./.lake/build/bin/vit-tiny-train
ViT-Tiny: 5526346 params
Generating train step MLIR...
  742145 chars
```

```
Compiling vmfbs...
  forward compiled
  eval forward compiled
  train step compiled
  session loaded
  train: 9469 images (256×256)
  5526346 params + m + v (63 MB)
training: 295 batches/epoch, batch=32, Adam, lr=0.000300,
          cosine, label_smooth=0.1, wd=1e-4
  no BN (LayerNorm only)
  step 0/295: loss=2.302585 (366ms)
Epoch 10/80: loss=1.552973 lr=0.000298 (102247ms)
  val accuracy: 2121/3904 = 54.33%
Epoch 20/80: loss=1.384668 lr=0.000275 (100662ms)
  val accuracy: 2258/3904 = 57.84%
Epoch 40/80: loss=1.178420 lr=0.000172 (102352ms)
  val accuracy: 2642/3904 = 67.67%
Epoch 60/80: loss=1.042742 lr=0.000054 (102175ms)
  val accuracy: 2779/3904 = 71.18%
Epoch 80/80: loss=0.986243 lr=0.000000 (102234ms)
  val accuracy: 2799/3904 = 71.70%
```

(Full log in `logs/vit_train.log`.)

Final val accuracy **71.70%** on Imagenette, wall time ∼**2.3 hours** — the fastest of any Imagenette-scale model in Part 1. Expanding the comparison table from the SE chapter:

Model	Params	MLIR	Step time	Total	Val acc
ResNet-34	21.29M	518 KB	1400 ms	9.5 h	90.29%
MobileNet V2	2.24M	741 KB	830 ms	5.4 h	87.09%
EfficientNet-B0	7.16M	938 KB	940 ms	6.2 h	87.58%
ViT-Tiny	5.53M	742 KB	360 ms	2.3 h	71.70%

ViT-Tiny is the fastest-per-step and the lowest-accuracy. That's the data-hunger effect: 9469 training images is tiny for a transformer. The ViT paper reached ImageNet-competitive accuracy only when trained on ImageNet-21K (14M images) or JFT-300M (300M images); at Imagenette scale the ConvNets' inductive bias (locality, translation equivariance) actively helps, and the ViT has to learn those properties from data it doesn't have.

That's not a problem with the framework, the proofs, or the ViT architecture — it's the right answer. A book that claimed "ViT always beats ResNet" would be lying. A book that reports the real numbers and says "ViT is a different scaling regime" is doing honest pedagogy. The same proof framework that covers ResNet and MobileNet and EfficientNet also covers ViT; the same training recipe runs on all of them. What's different is the *answer* you get, and why.

What Part 1 has established

Every layer primitive shipped in Part 1's trainers (MLP, CNN, ResNet, MobileNet, EfficientNet, ViT) has a machine-checked backward pass. The same `NetSpec` / `TrainConfig` / `.train` pipeline trains all of them with at most config-level changes. The MLIR each emits is between 20 KB (MLP) and 938 KB (EfficientNet-B0), and IREE compiles all of them the same way.

The bestiary in Part 2 then shows that *every other architecture you've seen in modern deep learning* — UNet, DETR, Mask R-CNN, DCGAN, CycleGAN, Pix2Pix, DDPM, Stable Diffusion, VAE, AlphaGo, AlphaZero, MuZero, Mamba, BERT, GPT, Whisper, CLIP, LLaVA, SAM, SegFormer, DeepLab v3+, Nyström-former, QANet, Evoformer — composes from the same small set of primitives plus a handful of architecture-specific bundled layers. Three rules (chain, additive fan-in, multiplicative fan-in). Five Jacobian tricks (diagonal, sparse Toeplitz, binary selection, rank-1 correction, outer product). 30 axioms, 45 theorems, zero sorrys.

That's the whole framework. The rest of the book is picking architectures off the shelf. Stamp collecting, as Rutherford would say.

Chapter 11

Bestiary of Architectures

The proof chapters above cover every layer shipped in Part 1's trainers (MLP, CNN, ResNet, MobileNet, EfficientNet, ViT). The **bestiary** is the book's Part 2: a catalogue of famous architectures expressed as pure `NetSpec` values — no training runs, no VJP commitments, just "here's what this architecture looks like in ∼15 lines of Lean."

Each bestiary entry introduces zero or one new `Layer` constructor for its architectural idiom. The primitive is bundled at the right abstraction level (a single Transformer encoder block, a whole Mamba block, a stage of Swin attention, etc.) — formalizing every sub-op of a modern architecture would be an asymptote we don't approach.

11.1 Bestiary-only `Layer` primitives

Twenty-two new `Layer` constructors were added by the bestiary chapters. None are codegen-backed (MlirCodegen emits `// UNSUPPORTED`); the goal is pedagogical shape + parameter accounting, not training runs.

Definition 82 (Mamba block)**.** Selective state-space block in the S6 formulation (Gu & Dao 2023). Signature: `mambaBlock (dim stateSize expand nBlocks : Nat)`. Bundles RMSNorm + linear expand + depthwise 1D conv + SiLU + selective-scan SSM + gated product + output projection, for `nBlocks` stacked layers. The selective scan is the novel primitive; everything else could be decomposed to existing layers if we cared to unpack the bundle.

Definition 83 (Swin Transformer stage)**.** Windowed multi-head self-attention at fixed spatial resolution (Liu et al. 2021). Signature: `swinStage (dim heads mlpDim windowSize nBlocks : Nat)`. Internal blocks alternate W-MSA and SW-MSA (shifted-window) to let information cross window boundaries.

Definition 84 (Patch merging)**.** Swin's 2×2 spatial downsample + linear channel projection (`inDim` → `outDim`). Transformer-side analog of a stride-2 conv.

Definition 85 (UNet encoder stage)**.** 2× `(conv3×3 + BN + ReLU)` then `maxPool-2`. Saves its pre-pool activation as a skip for the matching `unetUp`. Signature: `unetDown (ic oc : Nat)`.

Definition 86 (UNet decoder stage)**.** Transposed-conv 2× upsample + concat with matching skip + 2× `(conv3×3 + BN + ReLU)`. Signature: `unetUp (ic oc : Nat)`, where `oc` is both the output channel count and the expected skip width.

Definition 87 (Transformer decoder (DETR-style))**.** `nBlocks` blocks of self-attention over `nQueries` learned object queries + cross-attention against an encoder output + FFN. The query embedding is part of the layer's parameters. Signature: `transformerDecoder (dim heads mlpDim nBlocks nQueries : Nat)`.

Definition 88 (DETR prediction heads)**.** Per-query class head (linear `dim` → `nClasses+1` with the "no object" slot) + box head (3-layer MLP to 4 scalars `(cx, cy, w, h)`). Signature: `detrHeads (dim nClasses : Nat)`.

Definition 89 (ShuffleNet v1 stage)**.** Grouped 1×1 conv + channel-shuffle permutation + 3×3 depthwise + grouped 1×1 conv, for `nUnits` units (first downsampling, rest residual). The shuffle is parameter-free; grouping reduces 1×1 cost by g. Signature: `shuffleBlock (ic oc groups nUnits : Nat)`.

Definition 90 (ShuffleNet v2 stage)**.** `nUnits` v2 units (Ma et al. 2018). Basic unit (stride 1): channel-split $[X_1, X_2]$, leave X_1 alone, run X_2 through $1 \times 1 \to 3 \times 3$ DW $\to 1 \times 1$ at half-width, concat, then channel-shuffle. Downsample unit (stride 2): both branches see the full input; left does DW-3×3-stride-2 + 1×1, right does 1×1 + DW-3×3-stride-2 + 1×1; concat doubles channels. v2 throws out v1's grouped 1×1 convs (G2) and skip-add (G4) per the paper's practical guidelines. Signature: `shuffleV2Block (ic oc nUnits : Nat)`.

Definition 91 (Evoformer block (AlphaFold 2))**.** Dual-representation (MSA + pair) joint update: MSA row-attention with pair bias + MSA column-attention + MSA transition + outer-product-mean (→ pair) + triangle multiplicative (outgoing, incoming) + triangle self-attention (starting node, ending node) + pair transition. Signature: `evoformerBlock (msaChannels pairChannels nBlocks : Nat)`. The triangulation-aware operations are the key inductive bias.

Definition 92 (Structure Module (AlphaFold 2))**.** Recurrent Invariant Point Attention + backbone frame update + side-chain χ-angle prediction. **Weights shared across `nBlocks` rounds** — param count does not multiply by `nBlocks`. Signature: `structureModule (singleChannels pairChannels nBlocks : Nat)`.

Definition 93 (MobileViT block)**.** Hybrid local-conv + patch-level transformer (Mehta & Rastegari 2022). Local 3×3 conv → 1×1 projection to transformer dim → unfold into patches → L transformer blocks across patches → fold back → 1×1 projection back → concat with input → 3×3 fusion. Signature: `mobileVitBlock (ic dim heads mlpDim nTxBlocks : Nat)`. The unfold/fold operations are `pdiv_reindex`-style shape transformations; all the genuinely new math is already covered by the transformer proof chapter.

Definition 94 (ConvNeXt stage)**.** ConvNeXt residual block × `nBlocks`: 7×7 depthwise conv → LayerNorm → 1×1 expand (to $4c$) → GELU → 1×1 project (back to c) + residual. The transformer-era CNN block (Liu et al. 2022). Does not include downsampling; see `convNextDownsample`. Signature: `convNextStage (channels nBlocks : Nat)`.

Definition 95 (ConvNeXt downsampling)**.** Inter-stage spatial halving + channel-doubling: LayerNorm + 2×2 conv stride 2. Swin's `patchMerging` for CNNs. Signature: `convNextDownsample (ic oc : Nat)`.

Definition 96 (WaveNet block)**.** One stack of `nLayers` dilated causal residual blocks with doubling dilation rates $2^0, 2^1, \ldots, 2^{\texttt{nLayers}-1}$ (van den Oord et al. 2016). Each block: dilated 2-tap causal conv → gated activation $\tanh(\text{filter}) \odot \sigma(\text{gate})$ → 1×1 project back to `residualCh` (residual path) + 1×1 skip projection to `skipCh`. Skip outputs across blocks are summed into the final head. Signature: `waveNetBlock (residualCh skipCh nLayers : Nat)`. Output channels are `skipCh`: bestiary convention picks the skip path as the "forward" output since it's what feeds the final classifier.

Definition 97 (Positional encoding)**.** Sinusoidal frequency basis (Vaswani 2017, reused by NeRF 2020): $\gamma(p) = (\sin 2^0\pi p, \cos 2^0\pi p, \ldots, \sin 2^{L-1}\pi p, \cos 2^{L-1}\pi p)$. Zero trainable parameters — it's a deterministic lift of a low-dim coordinate into a high-frequency feature space where an MLP has enough wiggle room to represent sharp details. Output dim = inputDim ·2· numFrequencies. Signature: `positionalEncoding (inputDim numFrequencies : Nat)`.

Definition 98 (NeRF MLP core)**.** The whole NeRF network (Mildenhall et al. 2020) bundled as one primitive: 8 hidden ReLU-FC layers of `hiddenDim`, mid-skip concatenating $\gamma(x)$ at layer 5, dual output heads (1-dim volume density σ + 3-dim RGB via a direction-conditioned branch). Under 600K parameters at the canonical config. Signature: `nerfMLP (encodedPosDim encodedDirDim hiddenDim : Nat)`.

Definition 99 (Darknet residual block)**.** YOLOv3's Darknet-53 residual stack (Redmon & Farhadi 2018). `nBlocks` residual blocks at fixed `channels`, each being 1×1 conv ($c \to c/2$) + 3×3 conv ($c/2 \to c$) + residual add. Lighter than a standard ResNet bottleneck; heavier than a ResNet-18 basic block. Signature: `darknetBlock (channels nBlocks : Nat)`.

Definition 100 (Cross-Stage Partial block)**.** CSP (Wang et al. 2019), used by YOLOv4 onward. Splits input into two halves, processes one half through a stack of residual blocks, then concatenates with the untouched half and 1×1-projects to `oc`. The specific inner block varies across YOLO versions (C3 in v5, C2f in v8, C3k2 in v11); this primitive approximates all three at the same abstraction level. Signature: `cspBlock (ic oc nBlocks : Nat)`.

Definition 101 (Feature Pyramid Network)**.** Lin et al. 2017. Takes the four stage outputs of a CNN backbone (channels `c2`/`c3`/`c4`/`c5` at strides 4/8/16/32), projects each to `target` channels with a 1×1 lateral conv, merges them top-down (upsample-2× then elementwise add), and applies a 3×3 smoothing conv at each merged level. Output: four feature maps, each `target`-wide, at the original spatial resolutions.

Bundled because the cross-scale add doesn't fit a linear NetSpec. Standard kit in 2-stage detectors (Mask R-CNN, Cascade R-CNN) and single-stage detectors (RetinaNet). Signature: `fpnModule (c2 c3 c4 c5 target : Nat)`.

Definition 102 (Atrous Spatial Pyramid Pooling)**.** DeepLab v3+'s marquee module (Chen et al. 2018). Five parallel branches emitting `oc` channels each: (1) 1×1 conv, (2–4) 3×3 atrous convs at dilation rates 6 / 12 / 18, (5) global avg-pool + 1×1 conv + bilinear upsample. Concatenate, then a 1×1 fusion conv back to `oc`. All branches include BN + ReLU. Atrous rates widen the effective receptive field without changing param count; the pool branch supplies image-level context. Signature: `asppModule (ic oc : Nat)`.

Definition 103 (Inception module)**.** The GoogLeNet parallel-branch module (Szegedy et al. 2014). Four branches computed in parallel: 1×1 conv (b1 channels); 1×1 reduce then 3×3 (b2); 1×1 reduce then 5×5 (b3); 3×3 maxPool then 1×1 (b4). Concat along channels for `b1 + b2 + b3 + b4` outputs. The 1×1 dimension reducers on branches 2 and 3 are the paper's trick — they make the expensive 3×3 and 5×5 convs operate on reduced channel counts. Signature: `inceptionModule (ic b1 b2reduce b2 b3reduce b3 b4 : Nat)`.

11.2 Bestiary entries

The entries are grouped by task domain. The first block (*vision classifiers*) is where Part 1's VJP'd primitives show up at real-world scale — every layer is one you've already seen proved correct. Subsequent blocks step out to detection, segmentation, reinforcement learning, and the non-vision outliers (language, audio, 3D, multimodal, science).

11.2.1 Vision classifiers — Part 1's primitives at scale

Image-classification backbones built out of conv / pool / batch-norm / residual / attention / patch-embed — the exact layer kit VJP'd in Part 1. If a chapter of Part 1 proved it, a bestiary entry below puts it to work.

LeNet (`Bestiary/LeNet.lean`)

Zero new primitives — just `conv2d` + `maxPool` + `dense`. The 1998 original (LeCun et al. 1998, IEEE 1998). Variants: LeNet-5 (61K params, the canonical CNN) and LeNet-300-100 (266K params, pure-MLP baseline). Historical importance; still the ground-truth pattern every later CNN riffs on.

```
def leNet5 : NetSpec where
  name   := "LeNet-5"
  imageH := 32
  imageW := 32
  layers := [
    .conv2d 1 6 5 .valid .relu,          -- 32×32 → 28×28
    .maxPool 2 2,                        -- 28×28 → 14×14
    .conv2d 6 16 5 .valid .relu,         -- 14×14 → 10×10
    .maxPool 2 2,                        -- 10×10 → 5×5
    .flatten,
    .dense (16 * 5 * 5) 120 .relu,       -- 400 → 120
    .dense 120 84 .relu,
    .dense 84 10 .identity               -- 10-class MNIST output
  ]
```

AlexNet (`Bestiary/AlexNet.lean`)

Zero new primitives — five convs, three FCs, pools. The 2012 ImageNet winner that restarted modern deep learning (Krizhevsky et al. 2012, NeurIPS 2012). Variants: AlexNet (62M, paper-exact at 60M) + tiny CIFAR fixture. ∼58M of the 62M live in the three FC layers, which is precisely why every post-2015 CNN dropped FC stacks for `globalAvgPool` + one final dense. LRN is omitted (replaced in the field by BatchNorm in 2015); dropout is training-time only.

```
def alexNet : NetSpec where
  name   := "AlexNet (Krizhevsky 2012)"
  imageH := 227
```

```
  imageW := 227
  layers := [
    .convBn 3 96 11 4 .same,              -- 11×11 stride 4
    .maxPool 2 2,                         -- 3×3 stride 2 in paper
    .conv2d 96 256 5 .same .relu,
    .maxPool 2 2,
    .conv2d 256 384 3 .same .relu,
    .conv2d 384 384 3 .same .relu,
    .conv2d 384 256 3 .same .relu,
    .maxPool 2 2,
    .flatten,
    .dense (6 * 6 * 256) 4096 .relu,      -- ~95% of params live here
    .dense 4096 4096 .relu,
    .dense 4096 1000 .identity
  ]
```

SqueezeNet (`Bestiary/SqueezeNet.lean`)

Zero new primitives — the `.fireModule` constructor was already in `Types.lean`. AlexNet-level accuracy in 1.25M params via the fire module (Iandola et al. 2016, arXiv:1602.07360): squeeze 1×1 conv followed by parallel expand 1×1 + 3×3 convs concatenated. The early efficiency-CNN family alongside MobileNet and ShuffleNet. Variants: SqueezeNet 1.0 (1.25M, paper-exact), 1.1 (1.24M — earlier downsample), tiny fixture.

```
def squeezeNet1_0 : NetSpec where
  name   := "SqueezeNet 1.0"
  imageH := 224
  imageW := 224
  layers := [
    .convBn 3 96 7 2 .same,                    -- stem
    .maxPool 2 2,
    .fireModule 96  16 64  64,                 -- Fire2  → 128
    .fireModule 128 16 64  64,                 -- Fire3
    .fireModule 128 32 128 128,                -- Fire4  → 256
    .maxPool 2 2,
    .fireModule 256 32 128 128,                -- Fire5
    .fireModule 256 48 192 192,                -- Fire6  → 384
    .fireModule 384 48 192 192,                -- Fire7
    .fireModule 384 64 256 256,                -- Fire8  → 512
    .maxPool 2 2,
    .fireModule 512 64 256 256,                -- Fire9
    .conv2d 512 1000 1 .same .relu,            -- 1×1 to classes
    .globalAvgPool                             -- GAP emits 1000 logits
  ]
```

Inception v1 / v3 / v4 (`Bestiary/Inception.lean`)

One new primitive (§ 103). GoogLeNet's parallel multi-scale conv-branches concatenated along channels (Szegedy et al. 2014, arXiv:1409.4842); the 1×1 dimension reducer was invented here. Variants: GoogLeNet (7M, paper-exact), Inception-v3 (23.8M, paper 23M), Inception-v4 (33M, paper 42M — a bit low because our unified module approximates v3/v4's richer module catalog). Auxiliary classifiers (v1) and asymmetric factorizations (v3/v4) are bestiary omissions.

```
def googLeNet : NetSpec where
  name   := "GoogLeNet (Inception v1)"
  imageH := 224
  imageW := 224
  layers := [
    -- Stem
    .convBn 3 64 7 2 .same,
    .maxPool 2 2,                              -- 56×56
    .convBn 64 64 1 1 .same,
    .convBn 64 192 3 1 .same,
    .maxPool 2 2,                              -- 28×28
    -- Inception 3a, 3b
```

```
    .inceptionModule 192 64 96 128 16 32 32,    -- → 256
    .inceptionModule 256 128 128 192 32 96 64,  -- → 480
    .maxPool 2 2,                               -- 14×14
    -- Inception 4a..4e
    .inceptionModule 480 192 96  208 16 48  64,  -- 512
    .inceptionModule 512 160 112 224 24 64  64,  -- 512
    .inceptionModule 512 128 128 256 24 64  64,  -- 512
    .inceptionModule 512 112 144 288 32 64  64,  -- 528
    .inceptionModule 528 256 160 320 32 128 128, -- 832
    .maxPool 2 2,                                -- 7×7
    -- Inception 5a, 5b
    .inceptionModule 832 256 160 320 32 128 128, -- 832
    .inceptionModule 832 384 192 384 48 128 128, -- 1024
    .globalAvgPool,
    .dense 1024 1000 .identity
  ]
```

Xception (`Bestiary/Xception.lean`)

Zero new primitives — reuses existing `.separableConv`. "Extreme Inception" (Chollet 2017, arXiv:1610.02357): every conv is a depthwise-separable conv. The design choice that made MobileNet possible a year later. Variants: Xception (21.9M, paper-exact at 22M), tiny fixture. Residual skips around each block-of-three-sep-convs are implicit in the linear NetSpec.

```
def xception : NetSpec where
  name   := "Xception"
  imageH := 299
  imageW := 299
  layers := [
    -- Entry flow
    .convBn 3 32 3 2 .same,
    .convBn 32 64 3 1 .same,
    .separableConv 64 128 1, .separableConv 128 128 1,
    .maxPool 2 2,
    .separableConv 128 256 1, .separableConv 256 256 1,
    .maxPool 2 2,
    .separableConv 256 728 1, .separableConv 728 728 1,
    .maxPool 2 2,
    -- Middle flow: 8× (3 sep-convs + implicit residual), linearized
    .separableConv 728 728 1, .separableConv 728 728 1, .separableConv 728 728 1,
    .separableConv 728 728 1, .separableConv 728 728 1, .separableConv 728 728 1,
    .separableConv 728 728 1, .separableConv 728 728 1, .separableConv 728 728 1,
    .separableConv 728 728 1, .separableConv 728 728 1, .separableConv 728 728 1,
    .separableConv 728 728 1, .separableConv 728 728 1, .separableConv 728 728 1,
    .separableConv 728 728 1, .separableConv 728 728 1, .separableConv 728 728 1,
    .separableConv 728 728 1, .separableConv 728 728 1, .separableConv 728 728 1,
    -- Exit flow
    .separableConv 728 728  1,
    .separableConv 728 1024 1,
    .maxPool 2 2,
    .separableConv 1024 1536 1,
    .separableConv 1536 2048 1,
    .globalAvgPool,
    .dense 2048 1000 .identity
  ]
```

ShuffleNet (`Bestiary/ShuffleNet.lean`)

One new primitive (§ 89). Grouped 1×1 convs + channel shuffle (Zhang et al. 2017, arXiv:1707.01083). Variants: ShuffleNet 0.5× / 1.0× / 2.0× at $g = 3$, plus a tiny fixture.

```
def shuffleNet1x : NetSpec where
  name   := "ShuffleNet 1.0× (g=3)"
  imageH := 224
```

```
  imageW := 224
  layers := [
    .convBn 3 24 3 2 .same,                    -- stem
    .maxPool 2 2,
    .shuffleBlock 24  240 3 4,                 -- stage 2
    .shuffleBlock 240 480 3 8,                 -- stage 3
    .shuffleBlock 480 960 3 4,                 -- stage 4
    .globalAvgPool,
    .dense 960 1000 .identity
  ]
```

ShuffleNet v2 (`Bestiary/ShuffleNetV2.lean`)

One new primitive (§ 90). The efficient-CNN paper (Ma et al. 2018, arXiv:1807.11164) that called out FLOPs as a bad latency proxy: measured memory-access cost (MAC) directly and derived four practical guidelines (equal channel widths, avoid grouped convs, avoid fragmentation, avoid element-wise ops). v2's architecture throws out everything in v1 that violated those rules — no grouped 1×1 convs, no skip-add — in favor of channel-split + identity + concat + channel-shuffle. Variants: 0.5× (1.37M, paper 1.4M), 1.0× (2.29M, paper 2.3M), 1.5× (3.52M, paper 3.5M), 2.0× (7.41M, paper-exact), tiny fixture. All widths within 2% of paper.

```
def shuffleV2_1_0 : NetSpec where
  name   := "ShuffleNet v2 1.0×"
  imageH := 224
  imageW := 224
  layers := [
    .convBn 3 24 3 2 .same,                    -- stem
    .maxPool 2 2,
    .shuffleV2Block  24 116 4,                 -- stage 2
    .shuffleV2Block 116 232 8,                 -- stage 3
    .shuffleV2Block 232 464 4,                 -- stage 4
    .conv2d 464 1024 1 .same .relu,            -- 1×1 head conv
    .globalAvgPool,
    .dense 1024 1000 .identity
  ]
```

MobileViT (`Bestiary/MobileViT.lean`)

One new primitive (§ 93); reuses existing `invertedResidual` for the MV2 stages. Hybrid mobile backbone (Mehta & Rastegari 2022, arXiv:2110.02178): MobileNet V2 body with MobileViT blocks replacing some of the deeper inverted-residual stages. Variants: MobileViT-S (5.6M, paper-exact), XS (2.3M), XXS (1.3M), tiny fixture.

```
def mobileViTS : NetSpec where
  name   := "MobileViT-S"
  imageH := 256
  imageW := 256
  layers := [
    .convBn 3 16 3 2 .same,                    -- stem
    .invertedResidual 16 32 4 1 1,             -- stage 1: MV2
    .invertedResidual 32 64 4 2 3,             -- stage 2: 3× MV2 + downsample
    -- Stage 3: MV2 downsample + MobileViT (d=144, L=2)
    .invertedResidual 64 96 4 2 1,
    .mobileVitBlock 96 144 4 288 2,
    -- Stage 4: MV2 downsample + MobileViT (d=192, L=4)
    .invertedResidual 96 128 4 2 1,
    .mobileVitBlock 128 192 4 384 4,
    -- Stage 5: MV2 downsample + MobileViT (d=240, L=3)
    .invertedResidual 128 160 4 2 1,
    .mobileVitBlock 160 240 4 480 3,
    .conv2d 160 640 1 .same .relu,             -- 1×1 expansion
    .globalAvgPool,
    .dense 640 1000 .identity
  ]
```

ConvNeXt (`Bestiary/ConvNeXt.lean`)

Two new primitives (§ 94, § 95). ResNet-50 modernized with transformer design choices (Liu et al. 2022, arXiv:2201.03545): 7×7 depthwise + inverted bottleneck + LN + GELU + separate inter-stage downsamples. Proof-by-construction that CNNs didn't need to die in 2021. Variants: T (28M), S (50M), B (89M), L (198M), tiny fixture — all matching the paper within 1%.

```
def convNextT : NetSpec where
  name   := "ConvNeXt-T"
  imageH := 224
  imageW := 224
  layers := [
    .convBn 3 96 4 4 .same,                    -- patchify stem: 4×4 stride 4
    .convNextStage 96 3,                       -- stage 1: 3 blocks
    .convNextDownsample 96 192,
    .convNextStage 192 3,                      -- stage 2: 3 blocks
    .convNextDownsample 192 384,
    .convNextStage 384 9,                      -- stage 3: 9 blocks (the deep one)
    .convNextDownsample 384 768,
    .convNextStage 768 3,                      -- stage 4: 3 blocks
    .globalAvgPool,
    .dense 768 1000 .identity
  ]
```

Swin Transformer (`Bestiary/SwinT.lean`)

Two new primitives (§ 83, § 84). Variants: Swin-T / -S / -B / tiny. Swin-T lands at 28M params matching the paper (Liu et al. 2021, arXiv:2103.14030) exactly.

```
def swinT : NetSpec where
  name   := "Swin-T"
  imageH := 224
  imageW := 224
  layers := [
    .patchEmbed 3 96 4 3136,                   -- 56×56 patches, dim 96
    .swinStage 96 3 384 7 2,                   -- stage 1: 2 blocks
    .patchMerging 96 192,                      -- 56×56×96 → 28×28×192
    .swinStage 192 6 768 7 2,                  -- stage 2: 2 blocks
    .patchMerging 192 384,                     -- 28×28×192 → 14×14×384
    .swinStage 384 12 1536 7 6,                -- stage 3: 6 blocks
    .patchMerging 384 768,                     -- 14×14×384 → 7×7×768
    .swinStage 768 24 3072 7 2,                -- stage 4: 2 blocks
    .globalAvgPool,
    .dense 768 1000 .identity
  ]
```

11.2.2 Object detection

Localize and classify. Detection heads are where the linear `NetSpec` shape starts to creak: multi-scale FPN outputs need a graph, not a list. The bestiary entries show the single-scale view and defer the multi-head refactor to the limitations discussion.

YOLO v1/v3/v5/v8/v11 (`Bestiary/YOLO.lean`)

v1 (Redmon et al. 2016, arXiv:1506.02640) uses *zero* new primitives (`conv2d` + `maxPool` + `flatten` + `dense`; the YOLO-ness lives in the loss and output reshape, not the architecture). v3 adds § 99 for the Darknet-53 body. v5/v8/v11 use § 100 for their CSPDarknet backbones. Variants: YOLOv1 (271M, paper-exact), fast, tiny, YOLOv3 (40M backbone), YOLOv5s/m (3M/8M single-scale), YOLOv8n/s (0.7M/2.9M single-scale), YOLOv11n/m (backbone only). Multi-scale FPN detection heads don't linearize — same skip-connection problem as UNet; all entries show a single-scale view.

```
def yolo : NetSpec where
  name   := "YOLOv1 (Redmon 2016)"
  imageH := 448
```

```
  imageW := 448
  layers := [
    .conv2d 3 64 7 .same .relu,
    .maxPool 2 2,
    .conv2d 64 192 3 .same .relu,
    .maxPool 2 2,
    -- Block 3: 1×1 reduce + 3×3 expand
    .conv2d 192 128 1 .same .relu,
    .conv2d 128 256 3 .same .relu,
    .conv2d 256 256 1 .same .relu,
    .conv2d 256 512 3 .same .relu,
    .maxPool 2 2,
    -- Block 4: four 1×1+3×3 pairs, then 1×1 + 3×3
    .conv2d 512 256 1 .same .relu, .conv2d 256 512 3 .same .relu,
    .conv2d 512 256 1 .same .relu, .conv2d 256 512 3 .same .relu,
    .conv2d 512 256 1 .same .relu, .conv2d 256 512 3 .same .relu,
    .conv2d 512 256 1 .same .relu, .conv2d 256 512 3 .same .relu,
    .conv2d 512 512 1 .same .relu,
    .conv2d 512 1024 3 .same .relu,
    .maxPool 2 2,
    -- Block 5: two 1×1+3×3 pairs + two more 3×3 convs
    .conv2d 1024 512 1 .same .relu, .conv2d 512 1024 3 .same .relu,
    .conv2d 1024 512 1 .same .relu, .conv2d 512 1024 3 .same .relu,
    .conv2d 1024 1024 3 .same .relu,
    .conv2d 1024 1024 3 .same .relu,
    -- Block 6: two 3×3 at 7×7
    .conv2d 1024 1024 3 .same .relu,
    .conv2d 1024 1024 3 .same .relu,
    -- Head: flatten + 2 FC → reshape to (7, 7, 2·5 + 20) = 1470
    .flatten,
    .dense (7 * 7 * 1024) 4096 .relu,
    .dense 4096 (7 * 7 * (2 * 5 + 20)) .identity
  ]
```

Mask R-CNN (`Bestiary/MaskRCNN.lean`)

One new primitive (§ 101). The canonical two-stage detector + instance segmentation reference (He et al. 2017, arXiv:1703.06870). Five architectural pieces: ResNet-FPN backbone, RPN for anchor-box proposals, ROI-Align for per-proposal feature extraction (an orchestration step, not a layer), a 2-layer FC box head for classification + bbox regression, and a 4-conv + transposed-conv mask head for per-class 28×28 masks. Shown as separate NetSpecs per head (SAM-style decomposition). Param totals: backbone+FPN 45.8M, box head 14.3M, mask head 2.6M, RPN 0.6M — ∼63M in total, matching paper. The FPN cross-scale add is the one thing that demanded a new bundled primitive; everything else reuses existing ResNet / conv / dense primitives. DETR is its end-to-end transformer-era cousin; Mask2Former is the DETR-style instance-segmentation successor.

```
def maskRCNNBackboneR101 : NetSpec where
  name   := "Mask R-CNN backbone (ResNet-101-FPN)"
  imageH := 800
  imageW := 800
  layers := [
    -- ResNet-101 stem + body (C2..C5)
    .convBn 3 64 7 2 .same,
    .maxPool 3 2,
    .bottleneckBlock 64   256  3  1,           -- C2
    .bottleneckBlock 256  512  4  2,           -- C3
    .bottleneckBlock 512  1024 23 2,           -- C4
    .bottleneckBlock 1024 2048 3  2,           -- C5
    -- FPN: combines C2-C5 into a 4-level feature pyramid at 256 channels
    .fpnModule 256 512 1024 2048 256
  ]
```

DETR (`Bestiary/DETR.lean`)

Two new primitives (§ 87, § 88; Carion et al. 2020, arXiv:2005.12872); reuses existing `bottleneckBlock` (ResNet backbone), `patchEmbed` (absorbs the DETR 1×1 channel reduce), and `transformerEncoder`. Variants: DETR-R50 (41M, paper-exact), DETR-R101 (60M), tiny.

```
def detrR50 : NetSpec where
  name   := "DETR-R50"
  imageH := 800
  imageW := 800
  layers := [
    -- ResNet-50 backbone (stem + 4 stages)
    .convBn 3 64 7 2 .same,
    .maxPool 2 2,
    .bottleneckBlock 64 256 3 1,                -- C2
    .bottleneckBlock 256 512 4 2,               -- C3
    .bottleneckBlock 512 1024 6 2,              -- C4
    .bottleneckBlock 1024 2048 3 2,             -- C5
    -- Channel projection (2048→256) + flatten spatial + pos embed
    .patchEmbed 2048 256 1 625,                 -- 25×25 = 625 tokens
    .transformerEncoder 256 8 2048 6,           -- 6 encoder blocks, 8 heads
    .transformerDecoder 256 8 2048 6 100,       -- 6 decoder blocks, 100 queries
    .detrHeads 256 91                           -- class (→92) + box (→4)
  ]
```

11.2.3 Semantic segmentation

Pixel-level labeling. Symmetric encoder/decoder with skip connections is the recurring pattern; UNet is the canonical instance, and the same shape later became the diffusion-model backbone.

UNet (`Bestiary/UNet.lean`)

Two new primitives (§ 85, § 86). Skip connections are implicit — the i-th `unetUp` from the bottom pairs with the i-th `unetDown` from the top. Variants: original (1-channel, 2-class), RGB, small, tiny. Original lands at 31M params matching Ronneberger (Ronneberger et al. 2015, arXiv:1505.04597).

```
def unet : NetSpec where
  name   := "UNet (original, grayscale → 2-class)"
  imageH := 512
  imageW := 512
  layers := [
    .unetDown 1   64,                           -- encoder stage 1
    .unetDown 64  128,                          -- encoder stage 2
    .unetDown 128 256,                          -- encoder stage 3
    .unetDown 256 512,                          -- encoder stage 4
    .convBn 512 1024 3 1 .same,                 -- bottleneck part 1
    .convBn 1024 1024 3 1 .same,                -- bottleneck part 2
    .unetUp 1024 512,                           -- decoder (skip: encoder 4)
    .unetUp 512 256,                            --         (skip: encoder 3)
    .unetUp 256 128,                            --         (skip: encoder 2)
    .unetUp 128 64,                             --         (skip: encoder 1)
    .conv2d 64 2 1 .same .identity              -- 1×1 conv to 2 classes
  ]
```

DeepLab v3+ (`Bestiary/DeepLabV3Plus.lean`)

One new primitive (§ 102). The pre-transformer segmentation workhorse (Chen et al. 2018, arXiv:1802.02611) — still deployed widely in remote sensing, medical imaging, and autonomous-driving perception pipelines. Two ideas: atrous (dilated) convolutions in the backbone's last stage (param-count-free receptive-field expansion) + an ASPP module for dense multi-scale context at the deepest feature resolution. The "+" in v3+ adds a lightweight decoder that upsamples the ASPP output 4× and concatenates a low-level skip from backbone stage 2. Variants: ResNet-101 backbone (59M, paper ∼63M), MobileNet v2 backbone (5.7M, paper ∼6M mobile variant), tinyDeepLab fixture. The ASPP skip-to-stage-2 in the decoder doesn't linearize cleanly; same hack as UNet / WaveNet use. SegFormer (next entry) argues "do ASPP's multi-scale context via a transformer pyramid"; different mechanism, same goal.

```
def deeplabv3plusResnet101 : NetSpec where
  name   := "DeepLab v3+ (ResNet-101 backbone)"
  imageH := 513
  imageW := 513
  layers := [
    .convBn 3 64 7 2 .same,                    -- stem
    .maxPool 3 2,
    -- ResNet-101 body; stage 4 uses stride 1 (atrous in real impl)
    .bottleneckBlock 64   256  3  1,
    .bottleneckBlock 256  512  4  2,
    .bottleneckBlock 512  1024 23 2,
    .bottleneckBlock 1024 2048 3  1,
    .asppModule 2048 256,                      -- ASPP: 5 branches + fusion
    -- Decoder (skip from stage 2 doesn't linearize; prose notes it)
    .conv2d 256 256 3 .same .relu,
    .conv2d 256 256 3 .same .relu,
    .conv2d 256 21 1 .same .identity           -- Pascal VOC: 21 classes
  ]
```

SegFormer (`Bestiary/SegFormer.lean`)

Zero new primitives. Semantic segmentation (Xie et al. 2021, arXiv:2105.15203) via a hierarchical transformer pyramid (MiT encoder: 4 stages of `.transformerEncoder` glued by `.patchMerging`) and a lightweight MLP decoder (a handful of `.dense` calls). The decoder stays trivially small across all B0–B5 encoder sizes — the design argument of the paper is precisely that a good pretrained transformer feature pyramid makes the segmentation head cheap. Variants: MiT-B0 encoder (2.6M, paper 3.7M), B2 (19M, paper 25M), B5 (61M, paper 82M), shared MLP decoder (1M single-scale approx), tiny fixture. Uniform ∼25% undercount across all encoder sizes because real MiT uses depthwise convs inside the FFN and overlapping patch embeddings at inter-stage transitions; our `.transformerEncoder` and `.patchMerging` approximate the shape without those details. Comparison to DeepLab v3+: SegFormer's decoder is ∼3M params across all sizes, DeepLab's ASPP module is ∼15M and needs per-receptive-field tuning.

```
def segformerB2 : NetSpec where
  name   := "SegFormer MiT-B2 encoder"
  imageH := 224
  imageW := 224
  layers := [
    -- Stage 1: 224 → 56, 64 channels, 3 blocks
    .patchEmbed 3 64 4 (56 * 56),
    .transformerEncoder 64 1 256 3,
    -- Stage 2: 56 → 28, 128 channels, 4 blocks
    .patchMerging 64 128,
    .transformerEncoder 128 2 512 4,
    -- Stage 3: 28 → 14, 320 channels, 6 blocks
    .patchMerging 128 320,
    .transformerEncoder 320 5 1280 6,
    -- Stage 4: 14 → 7, 512 channels, 3 blocks
    .patchMerging 320 512,
    .transformerEncoder 512 8 2048 3
  ]
```

SAM (`Bestiary/SAM.lean`)

Zero new primitives. Promptable segmentation (Kirillov et al. 2023, arXiv:2304.02643): a ViT image encoder runs once per image, a tiny prompt encoder tokenizes clicks / boxes / masks, a lightweight transformer mask decoder cross-attends between image tokens, prompt tokens, and a handful of learned output queries. Image encoder is `.patchEmbed` + `.transformerEncoder` (the ViT kit); mask decoder is a small `.transformerDecoder` (from DETR, 4 queries, 2 blocks). Variants: SAM ViT-B encoder (88M, paper total 91M), ViT-L (307M, paper 308M), ViT-H (635M, paper 636M), shared 3.3M mask decoder, tinySAM fixture. The image encoder accounts for ∼99% of each variant's parameter budget, which is why EfficientSAM (Xiong et al. 2023) focused its distillation there.

```
def samEncoderH : NetSpec where
  name   := "SAM ViT-H image encoder"
```

```
  imageH := 1024
  imageW := 1024
  layers := [
    -- 16×16 patches on 1024×1024 → 64×64 = 4096 tokens
    .patchEmbed 3 1280 16 4096,
    .transformerEncoder 1280 16 5120 32          -- 32 blocks, dim 1280
  ]
```

11.2.4 Image generation

Networks whose output is a novel image. Backbones overlap heavily with segmentation (UNet again), but the training objective is generative: denoise a random input until it becomes a plausible sample.

VAE (`Bestiary/VAE.lean`)

Zero new primitives. The classical variational autoencoder (Kingma & Welling 2013, arXiv:1312.6114): encoder outputs $(\mu, \log \sigma^2)$ (represented as a single tensor with doubled final width), the reparameterization trick $z = \mu + \sigma \odot \epsilon$ samples z in training code, the decoder reconstructs. KL divergence between the learned latent distribution and a standard normal is the regularizer. Variants: MNIST MLP VAE (20-dim latent, textbook example), CIFAR conv VAE ($4 \times 4 \times 4$ spatial latent, SD-style), tiny fixture. All shown as encoder + decoder pairs. Training-code details (the actual sampling step, the KL loss) live outside the NetSpec. The same architectural template scales up to Stable Diffusion's VAE (see `StableDiffusion.lean`); 2013's 20-dim MNIST latent and 2022's $64 \times 64 \times 4$ SD latent are the same idea at different scales. VQ-VAE (discrete-codebook variant) and β-VAE (scaled KL) are mentioned in the prose notes.

```
def mnistVAEEncoder : NetSpec where
  name   := "MNIST VAE encoder (MLP, 20-dim latent)"
  imageH := 28
  imageW := 28
  layers := [
    .flatten,
    .dense 784 400 .relu,
    -- Output is 2×20 = 40: first 20 are  , last 20 are log ².
    .dense 400 40 .identity
  ]
```

DCGAN (`Bestiary/DCGAN.lean`)

Zero new primitives. Deep Convolutional GAN (Radford et al. 2015, arXiv:1511.06434) — the paper that made GAN training reliably work. Eight design guidelines (strided convs instead of pooling, BN everywhere except G's output / D's input, no hidden FC layers, ReLU in G / LeakyReLU in D, Adam with specific hyperparams) that became the default for every GAN paper since. Three NetSpecs: noise projector (dense $100 \to 4 \times 4 \times 1024$, 1.65M), generator convs (11M, paper ∼12M), discriminator (11M, paper-exact). Transposed convs in G are approximated by standard convs of matching kernel and channels (same params, spatial doubling is forward-pass-only); same hack appears in VAE and Stable Diffusion entries. GAN training dynamics (mode collapse, equilibrium stability) are all training-procedure concerns living outside the NetSpec.

```
def dcganGenerator : NetSpec where
  name   := "DCGAN generator (64×64 RGB)"
  imageH := 4                                    -- starting spatial (post-projection)
  imageW := 4
  layers := [
    -- Each .convBn stands in for a 4×4 transposed conv doubling spatial;
    -- same params (ic × oc × k²), upsampling is a forward-pass detail.
    .convBn 1024 512 4 1 .same,                  --  4×4 →  8×8
    .convBn 512  256 4 1 .same,                  --  8×8 → 16×16
    .convBn 256  128 4 1 .same,                  -- 16×16 → 32×32
    .convBn 128   64 4 1 .same,                  -- 32×32 → 64×64
    .conv2d   64    3 4 .same .identity          -- RGB output (tanh in real)
  ]
```

Pix2Pix (`Bestiary/Pix2Pix.lean`)

Zero new primitives. The paired-data ancestor of CycleGAN, from the same lab 9 months earlier (Isola et al. 2017, arXiv:1611.07004). UNet generator (8 levels, ∼70M approx vs paper ∼54M — our `.unetDown` / `.unetUp` use 2 convs per level where Pix2Pix uses 1 strided conv, so we overcount) + PatchGAN discriminator (identical to CycleGAN's, 2.8M). Trained with GAN loss + L1 reconstruction — the L1 term is a direct supervision signal that exists only because pairs exist. When you don't have pairs you fall back to CycleGAN's cycle-consistency trick. Hardware context: 2016–2017, 54M UNet on 256×256 images meant batch size 1 on a GTX 1080 Ti, which is how InstanceNorm became the default normalizer in this lineage — it was what fit.

```
def pix2pixGenerator : NetSpec where
  name   := "Pix2Pix generator (8-level UNet, 256×256 RGB)"
  imageH := 256
  imageW := 256
  layers := [
    -- Encoder: 8 × unetDown, channels 64 → 128 → 256 → 512 → 512 ×4
    .unetDown 3    64, .unetDown 64  128,
    .unetDown 128 256, .unetDown 256 512,
    .unetDown 512 512, .unetDown 512 512,
    .unetDown 512 512, .unetDown 512 512,         -- bottleneck at 1×1×512
    -- Decoder: 8 × unetUp mirroring the encoder
    .unetUp 512 512, .unetUp 512 512,
    .unetUp 512 512, .unetUp 512 512,
    .unetUp 512 512, .unetUp 512 256,
    .unetUp 256 128, .unetUp 128  64,
    .conv2d 64 3 1 .same .identity                -- 1×1 to RGB (tanh in real)
  ]
```

CycleGAN (`Bestiary/CycleGAN.lean`)

Zero new primitives. Unpaired image translation (Zhu et al. 2017, arXiv:1703.10593): two generators $G : X \to Y$ and $F : Y \to X$ plus two PatchGAN discriminators, trained with adversarial loss + cycle consistency loss $\|F(G(x)) - x\|_1$. The cycle constraint is what makes unpaired training work — without it, G could mode-collapse every x to one target. Generator is a Johnson-style ResNet (Johnson et al. 2016, arXiv:1603.08155): convs down + 9 `.residualBlock`s at 256 channels + convs up (11.4M, paper ∼11M). Discriminator is a PatchGAN: 5 strided convs with a 70×70 receptive field, outputs an $N \times N$ grid of patch real/fake logits (2.8M, paper ∼2.8M). Four-network pattern shown as two specs (G and D); the other F and D_X are architecturally identical copies. The "one clever loss" does the work — the architecture is quite ordinary.

```
def cycleganGenerator : NetSpec where
  name   := "CycleGAN generator (9-block ResNet, 256×256 RGB)"
  imageH := 256
  imageW := 256
  layers := [
    .convBn 3 64 7 1 .same,                       -- 7×7 stem
    .convBn 64  128 3 2 .same,                    -- 256 → 128
    .convBn 128 256 3 2 .same,                    -- 128 →  64
    .residualBlock 256 256 9 1,                   -- 9 res blocks (bottleneck)
    .convBn 256 128 3 1 .same,                    -- upsample stand-in
    .convBn 128  64 3 1 .same,
    .conv2d 64 3 7 .same .identity                -- 7×7 to RGB
  ]
```

DDPM (`Bestiary/Diffusion.lean`)

Zero new primitives. The denoiser in Ho et al.'s DDPM (2020, arXiv:2006.11239) is a UNet — literally the same § 85 and § 86 Ronneberger shipped in 2015. Everything "diffusion" lives in the training loop: a forward noise schedule adds Gaussian noise over T steps, a reverse process learns to predict the added noise, sampling iterates the reverse from pure noise back to a clean image. None of that is a layer. The timestep conditioning MLP (sinusoidal → dense → dense) is shown as a standalone `NetSpec` that reuses `positionalEncoding` from NeRF. Variants: CIFAR config (32x32 backbone), 256x256 high-res config, tiny

fixture, timestep embed. Our simplified backbone undercounts vs paper (paper DDPM adds residual blocks, GroupNorm, attention-at-low-res, and per-block time-embedding projection on top of the UNet); the spec's value is showing the architectural shape, not an exact param match. The lesson is the same one CLIP and NeRF taught: the novelty lives in the training procedure, not in layer design.

```
def ddpmCifar : NetSpec where
  name   := "DDPM (CIFAR-10, backbone approx)"
  imageH := 32
  imageW := 32
  layers := [
    -- Encoder: 32 → 16 → 8 → 4
    .unetDown 3   128,                          -- 32 → 16
    .unetDown 128 256,                          -- 16 →  8
    .unetDown 256 256,                          --  8 →  4
    .convBn 256 256 3 1 .same,                  -- bottleneck
    .convBn 256 256 3 1 .same,
    -- Decoder: 4 → 8 → 16 → 32
    .unetUp 256 256, .unetUp 256 256, .unetUp 256 128,
    .conv2d 128 3 1 .same .identity             -- predict added noise (3-ch)
  ]
```

Stable Diffusion (`Bestiary/StableDiffusion.lean`)

Zero new primitives. The paper that made generative image models consumer-reachable (Rombach et al. 2022, arXiv:2112.10752). Two architectural moves over DDPM, each individually small: (a) *latent diffusion* — run the diffusion process on 64×64×4 VAE latents instead of 512×512 pixels, cutting spatial work ∼64×; (b) *text conditioning via cross-attention* — at each interior UNet resolution, insert a spatial transformer block that cross-attends from image tokens to CLIP text embeddings. Shown as six separate NetSpecs: VAE encoder, VAE decoder, CLIP text encoder (123M, matches CLIP ViT-L/14 exactly), UNet backbone (202M backbone approx; real SD 1.5 UNet is 865M, missing ∼650M is the interleaved cross-attention), an explicit spatial-transformer-block spec (uses `.transformerDecoder` with `nQueries = 0` — same primitive Whisper's decoder uses, same mechanism as DETR's decoder applied at image-feature resolution), and a tiny end-to-end fixture. Three components are pretrained and frozen during SD's main training (VAE + text encoder) or constrained-frozen (CLIP); only the UNet trains. SDXL scales the UNet to 2.6B; SD 3 switches the UNet for a DiT transformer. The latent-diffusion plus text-conditioning template is the same.

```
def sdUNet15 : NetSpec where
  name   := "SD 1.5 UNet denoiser (backbone approx)"
  imageH := 64                                  -- latent resolution, not pixel
  imageW := 64
  layers := [
    .conv2d 4 320 3 .same .identity,            -- stem (latent → channels)
    -- Encoder: 64 → 32 → 16 → 8
    .unetDown 320  640,
    .unetDown 640  1280,
    .unetDown 1280 1280,
    -- Bottleneck at 8×8 (real SD inserts a spatial transformer here)
    .convBn 1280 1280 3 1 .same,
    .convBn 1280 1280 3 1 .same,
    -- Decoder: 8 → 16 → 32 → 64
    .unetUp 1280 1280,
    .unetUp 1280 1280,
    .unetUp 1280 640,
    .conv2d 640 4 3 .same .identity             -- predict noise (4-ch latent)
  ]
```

11.2.5 Reinforcement learning

Two-headed (policy + value) networks wrapped in a self-play + MCTS outer loop. The architectural side is a stack of residual CNN blocks; the complexity lives in the outer loop, not the network.

AlphaGo (`Bestiary/AlphaGo.lean`)

Zero new primitives. The original 2016 Lee Sedol system (Silver et al. 2016, Nature). Three separate networks: a 13-layer conv *policy* network (∼3.9M) trained first on 30M human games then fine-tuned via self-play, a 13-layer conv + FC-head *value* network (∼4M), and a shallow linear *rollout* policy used inside MCTS for fast tree playouts. Input is 48 hand-crafted Go feature planes (liberties, ladder patterns, 3×3 stone arrangements, etc.). AlphaGo Zero (next entry) throws all of this out — raw board only, one two-headed net, self-play only — and plays better (5185 Elo vs 3140). The one-sentence lesson is written into every follow-up paper: features the network can learn on its own, it will.

```
def alphaGoPolicyNet : NetSpec where
  name   := "AlphaGo policy network"
  imageH := 19
  imageW := 19
  layers := [
    -- 13 conv layers: one 5×5 stem + twelve 3×3 at 192 channels
    .convBn 48 192 5 1 .same,                       -- 48 hand-crafted planes in
    .convBn 192 192 3 1 .same, .convBn 192 192 3 1 .same,
    .convBn 192 192 3 1 .same, .convBn 192 192 3 1 .same,
    .convBn 192 192 3 1 .same, .convBn 192 192 3 1 .same,
    .convBn 192 192 3 1 .same, .convBn 192 192 3 1 .same,
    .convBn 192 192 3 1 .same, .convBn 192 192 3 1 .same,
    .convBn 192 192 3 1 .same,
    -- 1×1 conv to per-position move logit (reshape + pass scalar downstream)
    .conv2d 192 1 1 .same .identity
  ]
```

AlphaZero (`Bestiary/AlphaZero.lean`)

Zero new primitives — `convBn` + `residualBlock` + `conv2d` + `dense`. Two-headed (policy + value) network (Silver et al. 2018, arXiv:1712.01815), expressed as two separate `NetSpec` values sharing the body in prose. Variants: AlphaGo Zero (Go), AlphaZero chess, tiny fixture.

```
def alphaGoZeroPolicy : NetSpec where
  name   := "AlphaGo Zero (policy head)"
  imageH := 19
  imageW := 19
  layers := [
    .convBn 17 256 3 1 .same,                       -- 17 raw-history planes → 256
    .residualBlock 256 256 19 1,                    -- 19 residual blocks (shared body)
    .conv2d 256 2 1 .same .identity,                -- policy head: 2 filters
    .flatten,
    .dense (2 * 19 * 19) 362 .identity              -- 361 board moves + pass
  ]
```

MuZero (`Bestiary/MuZero.lean`)

Zero new primitives — three ResNet-style networks (representation, dynamics, prediction) reusing `convBn` + `residualBlock` + `dense`. The architectural novelty is the three-network factoring, not any single layer type (Schrittwieser et al. 2020, arXiv:1911.08265). Five `NetSpec` values per variant (rep, dyn body, dyn reward head, pred policy, pred value). Variants: Go, Atari (representation only), tiny.

```
def muZeroGoRepresentation : NetSpec where
  name   := "MuZero Go - representation h"
  imageH := 19
  imageW := 19
  layers := [
    .convBn 17 256 3 1 .same,                       -- observation → hidden
    .residualBlock 256 256 16 1                     -- 16 ResBlocks (AlphaZero-shaped body)
  ]
```

11.2.6 Beyond vision

Architectures where the task domain is not a 2D image: language, audio, 3D scene reconstruction, multi-modal embedding, scientific. Several of these (NeRF, CLIP) have essentially *no* architectural novelty — the

interesting work lives in the data, loss, or training procedure, and the bestiary entry exists to make that point.

Mamba (`Bestiary/Mamba.lean`)

One new primitive (§ 82). Variants: Mamba-130M / 370M / 790M / tiny, matching Gu & Dao's (2023, arXiv:2312.00752) param counts within ∼5%.

```
def mamba130M : NetSpec where
  name   := "Mamba-130M"
  imageH := 2048                                  -- context length (L tokens)
  imageW := 1
  layers := [
    -- dim=768, state=16, expand=2, 24 blocks
    .mambaBlock 768 16 2 24,
    .dense 768 50280 .identity                    -- LM head (GPT-NeoX vocab)
  ]
```

BERT / RoBERTa (`Bestiary/BERT.lean`)

Zero new primitives — uses `.transformerEncoder`, same kit as ViT / DETR, plus `.dense vocab→dim` standing in for the token-embedding table (faithful param count, shape semantics cheat since a linear `NetSpec` can't express the $L \to L \times D$ lookup). Variants: BERT-base (109M, paper 110M), BERT-large (335M, paper 340M), RoBERTa-base (124M, paper 125M), RoBERTa-large (355M, paper-exact), tinyBERT fixture. The architectural lesson is that RoBERTa (Liu et al. 2019, arXiv:1907.11692) = BERT (Devlin et al. 2018, arXiv:1810.04805); all RoBERTa gains came from training procedure (dynamic masking, more data, bigger batches, dropped NSP) — none of which lives in the NetSpec.

```
def bertBase : NetSpec where
  name   := "BERT-base"
  imageH := 512                                   -- max context length
  imageW := 1
  layers := [
    -- Token-embedding lookup approximated as dense (vocab → D)
    .dense 30522 768 .identity,
    -- 12 encoder blocks, post-norm, GELU, mlpDim = 4·D
    .transformerEncoder 768 12 3072 12,
    -- [CLS] pooler (tanh applied downstream)
    .dense 768 768 .identity
  ]
```

GPT-1 / GPT-2 (`Bestiary/GPT.lean`)

Zero new primitives. GPT-1 (Radford et al. 2018, OpenAI TR) and GPT-2 (Radford et al. 2019, OpenAI TR) are decoder-only counterparts of BERT. Same `.transformerEncoder`, now read as a stack of decoder blocks with a causal attention mask (a training-time detail, not a parameter). No pooler; GPT-2 uses pre-norm instead of BERT's post-norm (zero-parameter swap). Weight tying: the LM head reuses the token-embedding matrix, so our `.dense vocab→D` stand-in already pays for both input and output sides. Variants: GPT-1 (116M, paper 117M), GPT-2 small (123M, paper 124M), medium (353M), large (772M, paper 774M), XL (1.56B, paper 1.5B), tinyGPT fixture. Reference implementation: Karpathy's `nanoGPT` (∼300 lines of PyTorch) targets GPT-2 small and is the canonical mental model.

```
def gpt2Small : NetSpec where
  name   := "GPT-2 small"                         -- the nanoGPT target
  imageH := 1024                                  -- context length
  imageW := 1
  layers := [
    -- Tied token embedding (also serves as LM head projection)
    .dense 50257 768 .identity,
    -- 12 decoder blocks, pre-norm (mask is attention-time, not a param)
    .transformerEncoder 768 12 3072 12
  ]
```

QANet (`Bestiary/QANet.lean`)

Zero new primitives. The 2018 reading-comprehension architecture (Yu et al. 2018, arXiv:1804.09541) that killed the BiLSTM for SQuAD-style tasks. Core contribution: an encoder block combining 4 depthwise-separable convs (local context) with a self-attention + FFN transformer block (global context) — the "conv + attention hybrid" shape MobileViT and ConvNeXt rediscovered in 2022. Expressed with primitives we already have: 4 `.separableConv 128 128 1` calls + 1 `.transformerEncoder 128 8 512 1` per block. Per-block count: ∼270K; the paper's model encoder stack (7 blocks) lands at ∼1.9M, repeated 3 times in the full architecture. The BiDAF- style context-query attention and character/word embedding tables are omitted from the spec; described in prose. QANet's headline number was **3–4× training speedup** over BiLSTM-based competitors, a clean example of hardware (parallelization of convs vs. sequential RNN roll-outs) forcing an architectural choice. BERT landing 7 months later ended the SQuAD-as-benchmark era, but QANet's hybrid shape lived on — showed up repeatedly in vision 4 years later.

```
def qanetEncoderBlock : NetSpec where
  name   := "QANet encoder block (4× sep-conv + self-attn + FFN)"
  imageH := 400                                  -- representative context length
  imageW := 1
  layers := [
    -- 4 depthwise-separable convs (local context, kernel 7 in paper)
    .separableConv 128 128 1,
    .separableConv 128 128 1,
    .separableConv 128 128 1,
    .separableConv 128 128 1,
    -- Self-attention + FFN + 2 LayerNorms (global context)
    .transformerEncoder 128 8 512 1
  ]
```

Nyströmformer (`Bestiary/Nystromformer.lean`)

Zero new primitives. An efficient-attention transformer (Xiong et al. 2021, arXiv:2102.03902) that replaces the $O(n^2)$ softmax attention with an $O(n)$ Nyström-approximation computed via m landmark tokens, $m \ll n$. Crucially, this is a *compute* change, not a parameter change: the $W_Q/W_K/W_V/W_O$ projections and the FFN are identical to a standard transformer. Our `.transformerEncoder` spec is therefore *identical* to BERT's at each scale — base lands at 108M (vs BERT-base 109M), large at 333M (vs BERT-large 335M). The entry's pedagogical value is prose-level: most of the 2020–2022 efficient-attention literature (Linformer, Performer, Longformer, BigBird, Reformer, FlashAttention) is parameter-identical to vanilla attention, and the bestiary can't differentiate them at the layer level. Nyströmformer is worth highlighting because the specific trick — a 1928 result from numerical linear algebra, routed through kernel-method literature in the early 2000s, finally surfacing in transformers — is a genuinely long arc for one math idea.

```
def nystromformerBase : NetSpec where
  name   := "Nyströmformer base (BERT-base-shaped, O(n) attention)"
  imageH := 4096                                 -- long context
  imageW := 1
  layers := [
    -- Architecturally identical to BERT-base; Nyström approximation
    -- lives inside the attention kernel, not the NetSpec.
    .dense 30522 768 .identity,
    .transformerEncoder 768 12 3072 12
  ]
```

WaveNet (`Bestiary/WaveNet.lean`)

One new primitive (§ 96). Dilated causal convolutions for raw audio sample prediction (van den Oord et al. 2016, arXiv:1609.03499): exponential receptive field, linear parameter growth. The foundation of neural TTS and PixelCNN. Variants: single stack (0.4M), 3-stack (paper setup, with a NetSpec-linearity approximation), music (single stack, 4.1M), tiny. One honest limitation: the residual-vs-skip dual-output pattern doesn't linearize cleanly, so the 3-stack variant uses a simplified channel-flow approximation.

```
def waveNet : NetSpec where
  name   := "WaveNet (single stack, speech)"
  imageH := 16000                                -- 1 second @ 16 kHz
```

```
  imageW := 1
  layers := [
    -- Input embedding: 256 mu-law bins → 32 residual channels
    .conv2d 256 32 1 .same .identity,
    -- One stack of 10 dilated residual blocks (dilations 2 ..2 )
    .waveNetBlock 32 512 10,
    -- Output head: skip-sum → 1×1 → 1×1, 256-way categorical
    .conv2d 512 256 1 .same .relu,
    .conv2d 256 256 1 .same .identity
  ]
```

Whisper (`Bestiary/Whisper.lean`)

Zero new primitives. Encoder-decoder transformer over log-mel spectrograms (Radford et al. 2022, arXiv:2212.04356). Encoder is `.transformerEncoder` on the 1500 post-stem audio tokens; decoder is `.transformerDecoder` (from DETR) with `nQueries = 0` — a clean seq2seq decoder with self-attn + cross-attn + FFN, text tokens coming from a separate `.dense vocab→D` stand-in tied to the LM head. Variants: tiny (7M enc, paper 39M total), base (19M enc, paper 74M), small (85M enc + 153M dec+emb = 238M vs paper 244M), medium (302M enc, paper 769M), large (629M enc, paper 1.55B total), plus a shared decoder spec and tinyWhisper fixture. Encoder and decoder are the same size per variant; adding them together recovers paper totals within 2–3%. The multitask interface (swap a prefix token to change language or switch between transcribe / translate) is prompt-engineering, not architecture — Whisper's architectural novelty is essentially zero.

```
def whisperSmall : NetSpec where
  name   := "Whisper small encoder"
  imageH := 1500                                  -- post-stem audio tokens (3000 / 2)
  imageW := 1
  layers := [
    -- 12 encoder blocks, dim 768, 12 heads, mlp 3072
    .transformerEncoder 768 12 3072 12
  ]
```

NeRF (`Bestiary/NeRF.lean`)

Two new primitives (§ 97, § 98). The "it's literally just an MLP" paper (Mildenhall et al. 2020, arXiv:2003.08934). Under 600K params at canonical config; the architectural novelty is nonexistent. What makes NeRF work is the positional encoding + the volumetric-rendering loss — both *outside* the network, not layers in it. Variants: canonical (593K), fast (167K hidden=128), tiny fixture.

```
def nerf : NetSpec where
  name   := "NeRF (canonical)"
  imageH := 1
  imageW := 1
  layers := [
    -- Positional encoding of (x,y,z): 3 coords × 2 × 10 freqs = 60-dim
    .positionalEncoding 3 10,
    -- 8-layer MLP with mid-skip + dual density/RGB heads.
    -- encodedDirDim = 2·2·4 = 16 (2D direction, 4 frequencies)
    .nerfMLP 60 16 256
  ]
```

CLIP (`Bestiary/CLIP.lean`)

Zero new primitives. Two textbook encoders glued together by a contrastive loss (Radford et al. 2021, arXiv:2103.00020): a ResNet-50 or ViT-B for images, a 12-layer causal transformer for text, each with a final linear projection to a shared embedding space. Variants: CLIP-RN50, CLIP-ViT-B/32 (151M, paper-exact), CLIP-ViT-L/14 (427M, paper-exact), tiny fixture. The architectural lesson from CLIP is identical to NeRF's: the novelty lives in data and objective, not in layer design.

```
def clipViTB32ImageEncoder : NetSpec where
  name   := "CLIP-ViT-B/32 (image encoder)"
  imageH := 224
  imageW := 224
  layers := [
```

```
    -- ViT-B/32: patch size 32 → 49 patches, dim 768, 12 blocks, 12 heads
    .patchEmbed 3 768 32 49,
    .transformerEncoder 768 12 3072 12,
    -- [CLS] token → shared 512-dim embedding space
    .dense 768 512 .identity
  ]
```

LLaVA / LLaVA-1.5 (`Bestiary/LLaVA.lean`)

Zero new primitives. The cleanest exhibit of the modern vision-language pattern (Liu et al. 2023, arXiv:2304.08485; LLaVA-1.5: arXiv:2310.03744): frozen CLIP ViT encoder + small MLP projector + (mostly) frozen LLaMA/Vicuna language model. Trained in two stages — projector-only pretrain, then joint instruction fine- tune. Shown as separate NetSpecs per component: vision encoder (303M, matching CLIP ViT-L/14), LLaVA-1 single-linear projector (4M), LLaVA-1.5 two-layer MLP projector (21M), LLM backbones at 7B and 13B. The LLM specs undercount real LLaMA by ∼23% because our `.transformerEncoder` uses a standard 2-projection FFN while LLaMA uses SwiGLU (3 projections); depth / width / heads still match. Key ratio: the projector is **0.3%** of total LLaVA-1.5 7B parameters — the entire interesting work of the model lives in 21M of trainable bolt-on between two pretrained backbones. BLIP-2, Flamingo, and every modern VLM demo generalize this template with fancier adapters (Q-Former, Perceiver resampler, gated xattn-dense), but the frozen-backbone-plus-adapter shape is the same.

```
def llava15Projector : NetSpec where
  name   := "LLaVA-1.5 projector (2-layer MLP)"
  imageH := 576                                  -- visual token count (24×24)
  imageW := 1
  layers := [
    -- The entire LLaVA architectural contribution: map CLIP's 1024-dim
    -- image tokens into the LLM's 4096-dim embedding space. ~21M params
    -- - roughly 0.3% of total LLaVA-1.5 7B. GELU between is zero-param.
    .dense 1024 4096 .identity,
    .dense 4096 4096 .identity
  ]
```

AlphaFold 2 Evoformer (`Bestiary/Evoformer.lean`)

Two new primitives (§ 91, § 92). Dual-representation (MSA + pair) doesn't fit a linear `NetSpec` cleanly; the bestiary shows the two bundled primitives and notes the limitation (Jumper et al. 2021, Nature). Variants: full (76M), mini, tiny.

```
def alphaFold2 : NetSpec where
  name   := "AlphaFold 2 (Evoformer + StructureModule)"
  imageH := 384                                  -- max residues (N_res)
  imageW := 1
  layers := [
    -- 48 blocks of dual-representation processing (MSA + pair).
    -- Each block bundles: MSA row-attn (w/ pair bias), MSA col-attn,
    -- MSA transition, outer-product mean → pair, triangle-mul updates
    -- (out + in), triangle self-attention (start + end), pair transition.
    .evoformerBlock 256 128 48,
    -- Structure Module: 8 recurrent IPA rounds (weights shared).
    .structureModule 384 128 8
  ]
```

Appendix A

Getting started

If you want to just see it run — train a real neural network, end to end, with the Lean → MLIR → IREE pipeline from Part 1 — this appendix is the smallest path from zero to a trained model. Three tracks are included:

1. **No-GPU Docker demo.** Train MNIST on CPU in ∼5 minutes. Zero setup beyond Docker. Good if you don't have a GPU and just want to confirm the pipeline works.
2. **Native install (CUDA or ROCm).** Train real image models (ResNet, MobileNet, EfficientNet, ViT) on your GPU. Requires an IREE build step but gives you the full book's training runs.
3. **Lean-only (no IREE).** Read, build, and verify the proofs without running any neural-network training. Covers everything in Part 1 at the proof level.

Track 1: No-GPU Docker demo

The repository ships a `Dockerfile` that builds a CPU-only MNIST demo in a multi-stage image. Pulls Lean 4, builds IREE's CPU runtime, compiles the MLP trainer, and bakes MNIST into the image. Once built, the final image is ∼300 MB and doesn't require a GPU or Python.

```
git clone https://github.com/brettkoonce/lean4-mlir.git
cd lean4-mlir
docker build -t lean4-mlir-demo .
docker run --rm lean4-mlir-demo
```

Output: the same three-layer MLP shown in Chapter 3 (784 → 512 → 512 → 10) trains for 12 epochs on real MNIST, converges to ∼97.9% test accuracy. First build takes ∼10 minutes (dominated by IREE CMake + Ninja); subsequent `docker run` invocations reuse the cached image.

This is the fastest way to verify the Lean-to-MLIR-to-IREE pipeline actually works on your hardware before committing to a larger setup.

Track 2: Native install (CUDA or ROCm)

For training the larger networks (ResNet-34, MobileNet v2/v3/v4, EfficientNet, ViT) you need a GPU-accelerated IREE runtime. Steps:

1. **Install Lean 4.** Uses `elan` (like `rustup` for Rust) to manage toolchain versions.

   ```
   curl https://raw.githubusercontent.com/leanprover/elan/master/elan-init.sh \
     -sSf | sh
   ```

2. **Build the IREE runtime** for your GPU backend. See `IREE_BUILD.md` in the repo for the full CMake invocation. Once built, the FFI shim in `ffi/` links against `libiree_runtime_unified.a` from your build tree.

3. **Fetch data.**

```
./download_imagenette.sh   # Imagenette 320px, ~300 MB preprocessed
```

4. **Build a trainer.**

   ```
   lake build resnet34-train
   ```

   Other available targets, in roughly book order: `mnist-mlp-train`, `mnist-cnn-train`, `cifar-cnn-train`, `cifar-bn-train`, `resnet50-train`, `mobilenet-v2-train`, `mobilenet-v3-train`, `mobilenet-v4-train`, `efficientnet-train`, `efficientnet-v2-train`, `vgg-train`, `vit-tiny-train`.

5. **Run, setting the backend env vars.** The FFI picks up `IREE_BACKEND` and `IREE_CHIP` at runtime.

   NVIDIA / CUDA:

   ```
   CUDA_VISIBLE_DEVICES=0 IREE_BACKEND=cuda IREE_CHIP=sm_86 \
     .lake/build/bin/resnet34-train
   ```

   (`sm_86` is Ampere — RTX 30-series, A100. Use `sm_89` for Ada/RTX 40-series, `sm_90` for Hopper/H100.)

   AMD / ROCm:

   ```
   HIP_VISIBLE_DEVICES=0 IREE_BACKEND=rocm IREE_CHIP=gfx1100 \
     .lake/build/bin/resnet34-train
   ```

   (`gfx1100` is the 7900 XT/XTX. Use `gfx90a` for the MI200 series, `gfx942` for MI300.)

   CPU fallback:

   ```
   IREE_BACKEND=llvm-cpu .lake/build/bin/resnet34-train
   ```

   (No chip argument. Slower than GPU but useful for debugging trainers without a working GPU setup.)

First-run compile is slow. IREE spends 10–15 minutes compiling a ResNet-sized training step to vmfb. Subsequent runs reuse the cached vmfb under `.lake/build/` unless you delete it. The first-compile is a one-time cost per architecture, per GPU target.

Shell wrapper. The repo ships `run.sh` which sets the right env vars and tees output to a log file. Usage: `./run.sh <trainer> [gpu] [backend]` (e.g. `./run.sh resnet34`; `./run.sh efficientnet-v2 1 cuda`).

Track 3: Proofs only (no IREE needed)

If you just want to read the book and verify the proofs, you can skip IREE entirely. The proofs don't depend on the runtime.

```
git clone https://github.com/brettkoonce/lean4-mlir.git
cd lean4-mlir
lake exe cache get             # pull ~5 GB of precompiled Mathlib
lake build LeanMlir.Proofs.MLP LeanMlir.Proofs.CNN \
           LeanMlir.Proofs.BatchNorm LeanMlir.Proofs.Residual \
           LeanMlir.Proofs.Depthwise LeanMlir.Proofs.SE \
           LeanMlir.Proofs.LayerNorm LeanMlir.Proofs.Attention
```

If the build succeeds, every theorem in the proof suite type-checked against Lean's kernel — that is the entire verification. Zero `sorry`s, every VJP formula proved or tied to a numerically verified axiom (see Appendix B).

To also run the numerical gradient checks on the axioms:

```
python3 LeanMlir/Proofs/check_axioms.py
```

Every axiom in the suite has a finite-difference test that compares its claimed Jacobian formula against the numerical derivative. The check typically completes in under a minute and reports pass/fail per axiom.

Common troubleshooting

- **`HAL device not found` on launch** — usually means the IREE runtime wasn't built with the requested backend's HAL driver. CPU fallback (`IREE_BACKEND=llvm-cpu`) is the safest diagnostic: if it works on CPU but not GPU, the problem is in your IREE build's HAL drivers, not in the Lean code.
- **`lake build` fails with Mathlib cache errors** — run `lake exe cache get` to populate the local cache. Without it, the first build compiles all of Mathlib from source (∼45 min). With it, Mathlib oleans are downloaded directly (∼30 seconds).
- **First IREE compile appears stuck** — it isn't. ResNet-34's full training-step compile takes 10–15 minutes. Watch RAM usage; if it's climbing steadily the compile is progressing. If you want a faster check, try the MNIST MLP first (compiles in ∼30 seconds).

Appendix B

On Verification

The proofs in this book compile with **zero `sorrys`**. Every VJP correctness theorem — dense layers, convolution, batch normalization, residual connections, depthwise convolution, squeeze-and-excitation, layer normalization, and self-attention — is machine-checked by Lean's type system. If it builds, it's correct.

But "it builds" is only one of three independent checks we rely on. Composition proofs catch math errors, finite-difference checks catch formula errors in the axioms themselves, and a parallel JAX pipeline catches codegen and implementation errors. Each layer fails in a different way, so overlaying them is the defense. This appendix walks through all three.

B.1 Axioms

The Verified VJP Proofs suite contains **81 numbered items across the proof chapters (10 axioms + 65 theorems + 6 definitions), plus 22 architecture definitions in the bestiary** (103 total) across 9 Lean files. This is a deep shrink from earlier drafts (30 axioms, 45 theorems, 22 definitions): the foundation flip elevated the calculus axioms (chain rule, sum rule, product rule, identity, reindex) to theorems proved from Mathlib's fderiv; several chapter-specific opaque forwards (`conv2d`, `maxPool2`, `depthwiseConv2d`, `geluScalar`) became concrete definitions whose Jacobians are derived; and the diff-threading branch closed out every remaining "provable but deferred" Jacobian (pdivMat row independence, BN inverse-stddev broadcast, softmax, softmax cross-entropy, GELU, row-wise softmax smoothness, all seven transformer-level composition chains). The 10 surviving axioms are all genuine non-smoothness or bundled-codegen boundaries, falling into four buckets:

- **Subgradient at non-smooth points** (3, all in Ch 3): the guarded pdiv *_relu* axiom plus its two existence shortcuts `relu_has_vjp` and `mlp_has_vjp`. ReLU is non-differentiable at zero; the deep-learning convention picks a one-sided subgradient and these axioms encode that choice.
- **Bundled input VJPs** (3): `conv2d`, `maxPool2`, and `depthwiseConv2d` each contribute one axiom asserting the input-side backward function. Their forward operators and their weight/bias gradients are now proved from the foundation; only the input-side VJPs stay axiomatic because of padding-boundary and argmax-routing conventions.
- **Bundled multi-head SDPA** (2): `mhsa_has_vjp_mat` wraps the per-head softmax-attention chain into one HasVJPMat, with `mhsa_layer_flat_diff` as its Differentiable sibling. Removing them would require a per-head reduction framework beyond Chapter 10's scope.
- **Patch-embedding interface** (2): `patchEmbed_flat_has_vjp` and its Differentiable sibling `patchEmbed_flat_diff` cover the ViT patch-embedding step (224×224 RGB into 196 16-d patches projected to 192-d tokens). Opaque-codegen boundary: the "unfold-as-conv-with-stride" proof needs spatial-rearrangement machinery from Ch 4 lifted into the matrix world.

Everything else — 65 theorems, 28 definitions — is composition over the foundation and these 10 axioms, glued by the structural rules (chain, additive fan-in, multiplicative fan-in, three-tensor extension) and the closed-form Jacobian tricks each chapter introduces. The book's central claim — "new architectures, no new calculus" — holds in its strongest form: five of the nine content chapters (Ch 2, Ch 5, Ch 6, Ch 8, Ch 9) add zero new axioms.

B.2 Finite-difference gradient checks

The script `LeanMlir/Proofs/check_axioms.py` runs 25 finite-difference gradient checks. For each, it perturbs the input by ε, compares the claimed VJP against the centered difference $(f(x+\varepsilon) - f(x-\varepsilon))/2\varepsilon$, and asserts agreement within tolerance (typical max-error $\sim 10^{-11}$ at $\varepsilon = 10^{-5}$ in float64).

The coverage is deliberately broad: 7 of the 10 surviving axioms are FD-tested directly — pdiv`_relu` at non-zero points, `mlp_has_vjp` by full-network chain-rule composition, the input VJPs of `conv2d` / `maxPool2` / `depthwiseConv2d`, the bundled `mhsa_has_vjp_mat` (per-head sdpa_back stacked over the head axis), and `patchEmbed_flat_has_vjp` — and the remaining 18 checks belt-and-suspender the *proved* Jacobian theorems (pdiv`_dense`, all four BN pieces, the softmax Jacobian, softmax cross-entropy, GELU, conv / depthwise weight and bias grads, and the three single-head SDPA Q/K/V backwards). Even though those formulas are proved symbolically, a numerical check confirms the formula in the proof matches the formula the prose claims.

The three axioms that aren't FD-tested are `relu_has_vjp` (a pure existence shortcut, redundant with pdiv`_relu` which is tested) and the two Differentiable siblings `mhsa_layer_flat_diff` and `patchEmbed_flat_diff` — smoothness claims, not Jacobian formulas, so FD has no purchase on them.

FD is cheap, easy to reason about, and tight enough for spot-checking formulas, but it struggles at non-smooth points (ReLU at zero, maxPool tiebreaks, GELU's derivative-sign transitions) where the limit definition breaks down, and it can't probe what the *compiled* code actually computes on the GPU — only what the formula says in Python. For those gaps and for end-to-end pipeline verification, the third layer takes over.

B.3 The JAX parallel pipeline

A separate Lean → JAX → XLA pipeline (`jax/Jax/Codegen.lean`, $\sim$ 1100 lines) produces an idiomatic JAX training script from the same `NetSpec` the primary Lean → StableHLO MLIR → IREE pipeline consumes. XLA is the compiler JAX uses to produce GPU code (the same backend that powers TensorFlow and other frameworks); IREE is its Lean-side counterpart. The two stacks are independent end to end — different codegen, different runtime, different kernels — which is why agreement between them is a meaningful cross-check. Running both from identical initial parameters on identical batches gives us a pair of trace files, one per stack, that should agree modulo float32 rounding if both pipelines compute the same math.

The agreement is very tight. For the MNIST MLP, step-1 losses agree to $\sim 2 \times 10^{-7}$ — float32 ULP — across the JAX-CPU-vs-IREE-ROCm comparison, and phase-3 IREE output is **bit-identical across AMD and NVIDIA hardware** at step 1. For the MNIST CNN with batch norm, step-1 agrees to $\sim 10^{-4}$, looser because variance reductions over $\sim$ 100k-element tensors amplify cross-compiler reduction-tree differences — both pipelines do correct math; they just sum it in different orders. Full results are committed to the repo as reproducible JSON-Lines traces; see `traces/CROSS_BACKEND_RESULTS.md`.

Layered on top of the end-to-end trace diff is a per-axiom differential test in `tests/vjp_oracle/`, which compares each Lean-proved backward pass against JAX's `value_and_grad` autodiff on a minimal one-step training run. Nine cases — dense, dense+ReLU, conv, conv+BN, conv+maxPool, residual, depthwise, SE, attention — each agree with JAX autodiff at 1–2 ULP of step-2 loss. Any future hand-derived VJP added to the Lean proof base can be validated by a one-step comparison against JAX, catching algebraic errors that FD would miss (sign flips, wrong contraction axis, swapped indices).

Together, the three layers catch three different kinds of bugs:

- **Formal proofs** fail if the math is wrong — but type-checking doesn't prove the axioms are physically right, only that the theorems follow from them.
- **FD gradient checks** fail if a bundled axiom's formula is wrong — but FD can't spot compiler bugs or tiebreak disagreements.
- **The JAX-parallel pipeline** fails if either compiler has a codegen bug, if the runtime drifts, or if the hand-derived VJP disagrees with JAX autodiff — the places formal proofs and FD can't see.

Each layer fails differently. Overlaying them is the guarantee.

Acknowledgments

Thanks to **Monica Omar** on Lean Zulip, who closed the 6-level `Finset.sum` commutation in the Conv2D VJP proof.