Quantization and Fast Inference

A practitioner's guide to efficient AI

Vivek Kalyanarangan

MEAP

MANNING

Quantization and Fast Inference

A practitioner's guide to efficient AI

Vivek Kalyanarangan

MEAP

MANNING

Quantization and Fast Inference

welcome

Thank you for purchasing the MEAP for *Quantization and Fast Inference*. To get the most out of this book, you'll want to be comfortable with Python and PyTorch, and have built and trained a few neural networks. Some exposure to GPU execution will help, but you don't need to be a kernel author. The book is written for ML engineers, infrastructure engineers, and applied researchers with roughly two to six years of experience. If you've shipped a model to a real system and felt the weight of its latency, memory footprint, or serving cost, you're in the right place.

Quantization has become one of the load-bearing techniques of modern AI. A 70B-parameter model in FP16 is an expensive thing to serve; the same model in 4-bit can fit on a single consumer GPU and run fast enough to hold a conversation. The gap between "this model is interesting" and "this model is deployable" is, more often than not, a quantization problem. And yet most of the material on the subject is scattered across arXiv papers, framework-specific tutorials, and blog posts that hand-wave the math and the numerical gotchas that actually bite you in production.

I wrote this book to fill that gap. Every technique in these pages is grounded in real measurements from real models running on real hardware — models you and I both use at work, not toy tensors. When you see a perplexity number, a throughput figure, or a memory breakdown, it comes out of a companion script that you can run and modify yourself. The Github repository that accompanies the book is where the prose meets the silicon: if something in the text makes you skeptical, you can re-run the experiment and check. I think this is the only honest way to write about quantization, because the field is full of claims that sound right and fall apart the moment you profile them.

The book builds up from the foundations — fixed-point arithmetic, affine mappings, the choices that govern how you map a continuous weight distribution onto a discrete grid — through the standard toolkit of post-training quantization and quantization-aware training, and into the techniques

that define the large language model era. You'll work through the formats that matter in practice (INT8, FP8, FP4, NF4, ternary), the calibration and fine-tuning recipes that make them work, the cross-framework export paths that get a quantized model out of research and into a runtime, and the deployment stacks where the speedups are actually realized. Along the way we'll confront the places where intuition misleads: why symmetric and asymmetric quantization behave differently for activations, why some "faster" formats are slower in practice, and why a model can pass every numerical check and still regress on downstream tasks.

The field moves quickly, and I'll update these chapters through the MEAP as new techniques stabilize and as your feedback shapes the book. Please post questions, corrections, and suggestions in the liveBook Discussion forum — a book like this gets sharper with every reader who pushes back on it.

—Vivek Kalyanarangan

In this book

1 Facing the Efficiency Wall

This chapter covers

- The memory-bandwidth bottleneck
- Why quantization targets the dominant cost
- The floating-point to integer transition

For most of the history of machine learning, efficiency was a secondary concern. Models were small enough to fit comfortably in memory. Inference was fast enough to feel instantaneous. When performance lagged, the usual remedies, such as better hardware, modest architectural tweaks, or more aggressive batching, were generally sufficient. Accuracy was the main currency, and the cost of getting there was often treated as an operational detail.

That era has ended.

The modern generation of models, especially large transformers, has pushed inference across a qualitative threshold. Parameter counts have exploded, context lengths have stretched by orders of magnitude, and workloads that once behaved like ordinary applications now behave like infrastructure. Latency flattens even on powerful GPUs. Utilization looks suspiciously low. Power draw and memory bandwidth, not arithmetic throughput, become the binding constraints.

Quantization is the technique of representing neural network weights and activations using fewer bits—typically moving from 16-bit or 32-bit floating point down to 8-bit or 4-bit integers. By reducing the number of bits that must be stored and moved through the memory hierarchy, quantization directly attacks the dominant cost of modern inference—data movement—directly targeting bytes moved per token without changing model structure or semantics. It reduces memory footprint, bandwidth, and energy in one stroke, and it composes cleanly with almost every other optimization you might already be using.

This book is for ML engineers, infrastructure teams, and applied researchers who need to deploy models in production. If you've stared at a GPU utilization dashboard wondering why your expensive hardware sits half idle, watched cloud bills climb faster than traffic, or tried to squeeze a model onto an edge device and failed, this book is for you. We assume familiarity with PyTorch and basic neural network concepts, but no prior background in numerical methods or hardware architecture.

Before we learn how to apply quantization to your infrastructure, we need a clear picture of why efficiency has become unavoidable, why quantization is the central lever, and what kind of tradeoffs you are about to navigate. That clarity will anchor every technical decision for the rest of the book.

Let's start by making the cost crisis concrete.

1.1 The cost crisis in memory, latency, and power

Since 2022, progress in LLMs has largely followed one blunt but effective strategy: make the model bigger and train it longer. That strategy worked spectacularly for capability—but it changed the economics of inference. According to data tracked by Our World in Data, the number of parameters in frontier AI models has approximately doubled every year since 2010. Context lengths expanded by four to sixteen times. Those multipliers don't add—they multiply.

Here is the punchline up front: *modern LLM systems are limited by memory movement and power, not by raw compute*. That single fact explains why GPUs sit underutilized, why latency plateaus even on faster hardware, and why cloud bills scale faster than traffic.

At inference time, a model is not "thinking." It is doing two very mundane things over and over: reading numbers from memory and performing simple arithmetic on them. The surprise for most people is not *what* happens—but *which of these dominates cost.*

Compute is cheap. Data movement is not

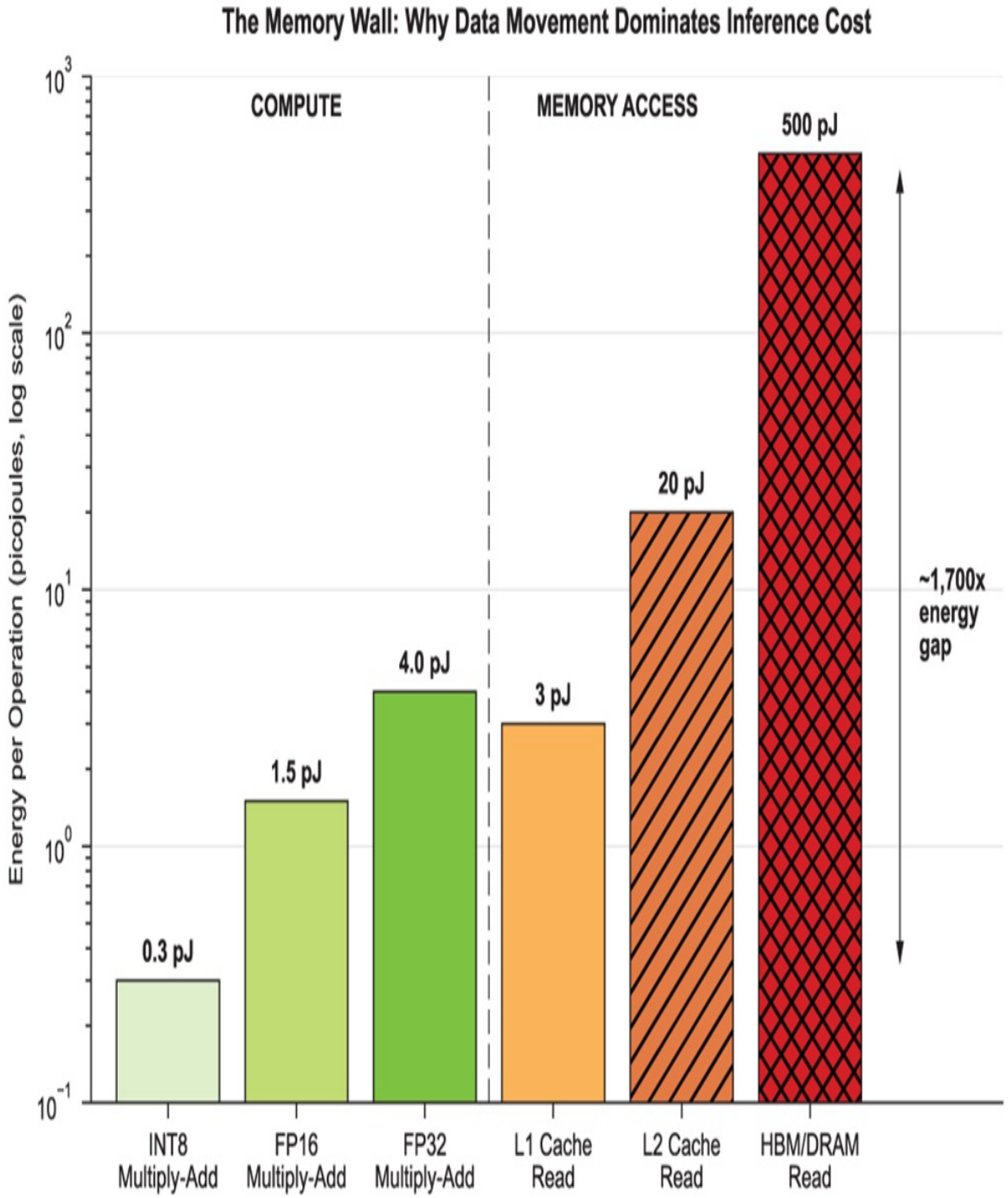
Before we go further, let's sanity-check the energy numbers you're about to see. These are order-of-magnitude, hardware-agnostic figures grounded in published computer-architecture measurements. Exact values vary by process node and vendor, but the *ratios* are stable across decades of silicon. The numbers below illustrate the approximate energy cost of common operations in modern hardware. Arithmetic operations such as multiply-adds require relatively little energy, while the cost of accessing memory increases dramatically as data moves farther from the processor—from on-chip caches to off-chip DRAM.

Approximate energy costs per operation: an FP32 fused multiply-add, a single multiply-then-add, the basic arithmetic step inside every matrix multiplication and convolution, consumes roughly 3–5 picojoules (pJ). The FP16 variant costs roughly 1–2 pJ, and the INT8 (8-bit integer) variant costs roughly 0.2–0.5 pJ. These figures come from measured ALU energy in modern CMOS designs (Horowitz, "Computing's Energy Problem and What We Can Do About It," ISSCC 2014, and follow-up industry surveys).

Approximate energy costs per memory access: an L1 cache access costs roughly 1–5 pJ, an L2 cache access roughly 10–30 pJ, and an HBM (High Bandwidth Memory, the stacked DRAM used on modern GPUs) or off-chip DRAM access roughly 300–1000 pJ.

You can see the numbers for yourself in figure 1.1, where we plot the energy required for each operation. Notice the clear wall between memory access and compute.

Figure 1.1 Fetching data from HBM costs roughly 1,700× more energy than an INT8 multiply-add. This gap explains why inference systems are bottlenecked by memory, not compute.



The exact values matter less than the ratio. *Fetching data from DRAM costs roughly two to three orders of magnitude more energy than arithmetic.* This gap is structural. Compute energy scales aggressively with process improvements; memory access energy does not.

A running example: serving a 7B-parameter model

Let's work with a model size that many teams actively deploy today: models like Llama 2 7B or Mistral 7B represent this class well. At 7 billion parameters, the raw model size varies by precision: at FP32, the model occupies roughly 28 GB (7 billion parameters times 4 bytes); at FP16, roughly 14 GB; at INT8, roughly 7 GB.

So far, this looks like a simple storage problem. It isn't. The real cost shows up when the model runs.

A note on 16-bit floats and the KV cache .

Three terms appear in nearly every memory calculation in this chapter, and they're worth pinning down once. The KV cache (key-value cache) stores the attention keys and values for every token generated so far. The model re-reads it at each step to maintain context, which is why its size grows linearly with sequence length. K and V refer to those key and value matrices individually. Hidden size is the width of each layer's internal representation: 4,096 for a typical 7B-parameter model.

The other recurring source of confusion is which 16-bit format we mean by "FP16" or "half precision." Two coexist in practice. IEEE FP16 uses 5 exponent bits and 10 mantissa bits, with a maximum representable value of about 65,504. BF16 uses 8 exponent bits and 7 mantissa bits, with a maximum value of roughly 3.4×10^{38}. BF16 keeps FP32's dynamic range at the cost of coarser precision and has become the default for LLM training since roughly 2020; when you load a Hugging Face checkpoint with torch_dtype=torch.bfloat16, you are getting BF16 weights. Both formats occupy 2 bytes per value, so every memory-traffic and energy calculation in this chapter applies identically regardless of which 16-bit format is in use.

For each generated token, a transformer moves data through memory in three places: the model's weights, the KV cache, and activations and intermediate buffers between layers. Together, these determine the energy cost of inference. For a realistic 7B-parameter transformer with a roughly 2,048-token context, the dominant contributors to memory traffic per generated

token are model weights at FP16 (roughly 14 GB), KV cache reads and writes (roughly 1.0 GB), and activations and intermediates (roughly 0.5 MB, negligible by comparison).

This already includes both K and V, FP16 storage, the model's 32 layers, its hidden size, and the full context length (because attention reads the entire cache on each new token). Taken together, this results in approximately 15 GB of memory traffic per generated token.

Using a conservative HBM energy estimate of 500 picojoules per 4 bytes transferred, this corresponds to roughly *1.9 joules per generated token*.

Scaling to real workloads

Now let's translate that into something operational. Assume a modest but realistic serving rate for one replica, a single running copy of the model behind a load balancer and the unit of capacity that operations teams scale up and down: 1,000 generated tokens per second. Using roughly 1.9 joules per token, the power draw is approximately 1.9 kW per replica.

This is steady-state power, not a peak. Run continuously for a year: 1.9 kW times 24 hours times 365 days equals roughly 16,600 kWh per year. For context, that is about 1.5× the annual consumption of a typical U.S. household (U.S. Energy Information Administration, 2023) from one replica.

In practice, no production system runs a single replica in isolation. Ten replicas draw roughly 19 kW continuously, comparable to a small commercial office. A hundred-replica deployment crosses 190 kW, in the load range of a mid-sized commercial building. A thousand-replica regional service approaches 2 MW of steady draw, which is the territory where utility interconnect agreements, not GPU procurement, become the binding constraint.

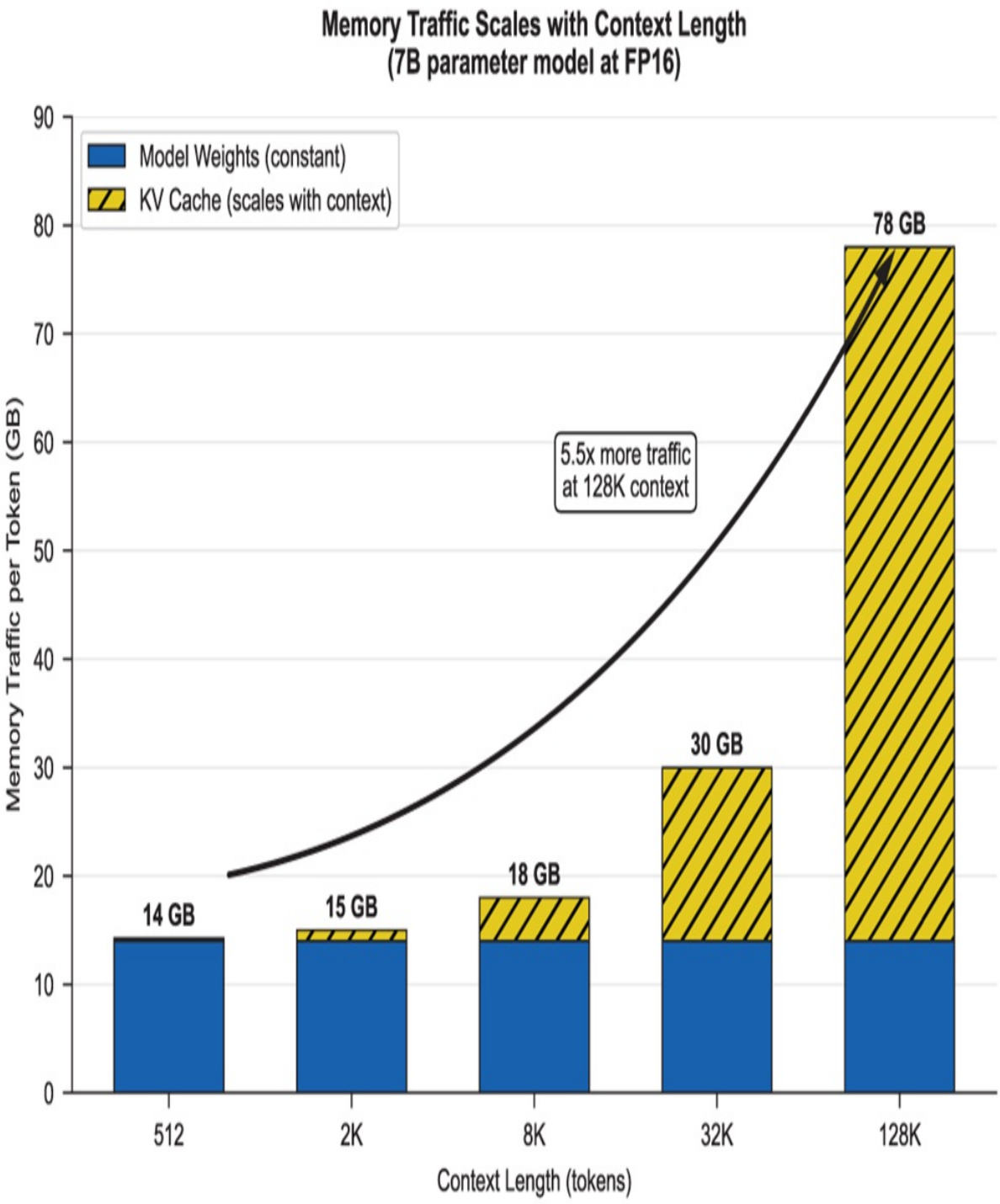
This is how inference crosses an invisible line, from an application concern to an infrastructure and energy-planning problem.

Three trends amplify everything we just calculated. First, models keep growing—parameters increase faster than memory bandwidth. Second,

context lengths keep expanding—the KV cache adds even more memory traffic. Third, test-time compute with reasoning models requires producing more tokens for the same query.

Figure 1.2 makes the second trend concrete by showing how memory traffic per token changes as context length scales from 512 tokens to 128K. The model weights remain fixed, but the KV cache grows linearly and quickly dominates total traffic.

Figure 1.2 Memory traffic per token as context length grows. Model weights (filled) stay constant, but the KV cache (hatched) scales linearly with context. At 128K tokens, total memory traffic reaches 78 GB per token—5.5× more than at 512 tokens.



Hardware improves, but power budgets, memory bandwidth, and thermal limits do not keep pace. The wall is not accidental; it is what you get when the dominant cost scales with bytes moved and your hardware roadmap scales with arithmetic throughput.

The first mental anchor of this book:

Throughout this book, we will pause at key moments to establish mental anchors, foundational ideas worth returning to when the details get complex. Here is the first: Bit precision is not primarily an accuracy decision; it is an energy decision. Every extra bit increases memory footprint, increases memory traffic, increases energy per token, and caps throughput long before compute does.

You've now seen the wall, but not yet the escape hatch. The natural next question is this: if lower precision saves this much energy, why doesn't it destroy model quality? Answering that carefully is the rest of this book.

1.2 Why quantization is the practical response

When faced with rising costs, the reflex is to wait for the next generation of hardware. More FLOPs. Faster GPUs. Wider memory buses. The analysis from section 1.1 already hints at why this instinct fails. Hardware progress overwhelmingly favors *compute density*, not *energy per byte moved*. Arithmetic units shrink and peak throughput grows rapidly, but the cost of fetching data from off-chip memory (any memory physically separate from the processor die, such as the HBM stacks on a GPU or conventional DRAM) improves far more slowly. Even high-bandwidth memory remains orders of magnitude more expensive, energetically, than a multiply-add. This is a structural mismatch, not a short-term inefficiency. Any response that does not *directly reduce memory traffic* is, at best, a partial solution.

Architectural changes such as *Mixture-of-Experts*, conditional computation, or sparse activation patterns reduce the amount of work performed per token. They can be highly effective, but they require architectural changes, complicate training and serving, and often shift cost rather than eliminate it. Most importantly, they are not drop-in: you cannot apply them to an existing deployed model without retraining or redesign.

Attention optimizations and kernel tricks, such as FlashAttention (an algorithm that restructures attention computation to minimize memory reads), fused kernels (GPU programs that combine multiple operations into a single

pass to avoid extra memory round-trips), and improved scheduling, dramatically reduce overheads inside attention blocks. They are essential optimizations, and they are already widely used. Still, these techniques optimize *how* data moves, not *how much* data must move. The model weights still need to be read. The KV cache still needs to be accessed. The dominant memory traffic remains.

Smarter batching and caching strategies, such as dynamic batching, speculative decoding, and request coalescing, improve throughput and tail latency under load. They are operationally critical, but they are amortization strategies, not reductions in per-token cost. When load rises, the same underlying energy bill eventually asserts itself.

Knowledge distillation trains a smaller "student" model to reproduce the outputs of a larger "teacher," compressing learned behavior into fewer parameters. It is out of scope here for one reason: distillation produces a *different model*, not a different representation of the same one, and that crosses into a different design space than the rest of this book covers.

Quantization is different in one crucial way: it scales down the dominant term linearly. Reducing precision from 16 bits to 8 bits cuts model weight size in half, memory bandwidth per token in half, and energy per token in half. The same is true when going to 4 bits or mixed-precision schemes. No architectural changes. No full-scale retraining required (except for some techniques that use tiny calibration datasets). No changes to the computation graph. The operations are the same. The *bytes are fewer*. This is why quantization appears everywhere in practical systems, from mobile inference to data-center LLM serving. It is not because it is theoretically elegant, but because it is brutally effective.

At this point, a reasonable objection arises: if precision drops so much, why doesn't everything break? The answer lies in what neural networks actually compute. Weights in a trained network do not encode exact values in the way a physics simulation does. They encode directions in a high-dimensional space (which features matter), correlations between inputs (which features co-occur), and relative influence (how strongly each connection contributes to the output). Most parameters tolerate small perturbations without materially changing the output. Neural networks are redundant, noisy by

design, and trained under stochastic approximations. Quantization does not introduce a fundamentally new kind of error. It introduces a bounded, structured approximation, one the network is often already equipped to absorb. This is why reduced precision degrades quality gradually rather than catastrophically.

In practice, the degradation is remarkably gentle at moderate bit-widths. As demonstrated by Frantar et al. in "GPTQ: Accurate Post-Training Quantization for Generative Pre-trained Transformers" (2023), weight-only INT4 quantization of Llama 2 7B using GPTQ adds only ~0.1 perplexity points on WikiText-2 (from 5.47 to 5.61), while INT8 is nearly indistinguishable from FP16. Below 4-bit, quality degrades more noticeably and method choice matters enormously: naive 3-bit quantization can cost several perplexity points, while specialized methods likeAWQ (Lin et al., 2023) orOmniQuant (Shao et al., 2023) narrow that gap significantly.

Another way to frame this is to recognize that inference already lives in an approximate world. Training noise, dropout, numerical nondeterminism, and finite batch statistics all introduce variation. Quantization adds one more controlled source of error. When done carefully, errors distribute across layers, biases can be corrected, and outliers can be handled explicitly. The result is not "wrong answers," but *slightly different answers*, often well within acceptable tolerances.

In practice, model quantization is a series of recurring decisions that all trade accuracy, performance, and deployment constraints against one another. You must decide what to quantize: weights, activations, or the key-value cache, and how to do it, whether per-tensor, per-channel, in groups, or in blocks. You must also decide when quantization happens: after training (post-training quantization, or PTQ), during training (quantization-aware training, or QAT), or through low-bit adaptation techniques like QLoRA.

Just as importantly, you must choose how far to push precision: 8-bit, 4-bit, mixed precision, or lower, and where the model will ultimately run, from GPUs and CPUs to edge devices and specialized runtimes.

There is no single correct configuration. Each choice reshapes the system's behavior and exposes different constraints. The purpose of working through

these decisions is not to memorize techniques, but to develop the judgment needed to reason about trade-offs and select the right approach for a given context.

The second mental anchor of this book:

Quantization works not because models are simple, but because they are redundant. Or said differently: the objective is not maximum precision, but sufficient precision at minimum energy.

You have now seen why quantization is the practical response. The next section shows what it actually means, numerically, to leave the floating-point world behind and enter the integer domain, where every value sits on a fixed grid.

1.3 Mapping Floating Point vs Integer

Before we talk about *how* quantization works, we need to understand what changes when we move from floating point to integer arithmetic, that is, from a number system where each value carries its own scale (like scientific notation) to one where all values share a fixed, uniform spacing. This is not merely a precision reduction, it is a shift between two fundamentally different ways of representing numbers, each with distinct tradeoffs for neural network inference.

Quantization is not just about using fewer bits. It is about crossing a boundary between two very different numerical and computational worlds: floating point and integer arithmetic.

1.3.1 Floating point's hidden costs

Floating point numbers were not designed for machine learning. They were designed for scientific computing. Their original goal was to represent numbers that vary across *enormous* ranges: astronomical distances, microscopic measurements, probabilities near zero, while preserving relative precision.

To do this, floating point representations combine a *mantissa* that stores significant digits and an *exponent* that scales those digits dynamically. The result is a system that automatically adjusts precision based on magnitude. Large numbers get coarse resolution. Small numbers get fine resolution. The spacing between representable values stretches and shrinks as needed. This flexibility is extraordinarily powerful. It is also expensive. Floating point optimizes for *numerical expressiveness*, the ability to represent a wide range of magnitudes with precision that adapts to each value's scale, not efficiency. That tradeoff made perfect sense when compute was scarce and memory traffic was not the dominant concern. In modern inference systems, the balance has flipped.

At inference time, models are no longer solving delicate numerical problems. They are applying learned transformations repeatedly and predictably. Yet floating point brings along the full machinery it was designed for: variable precision (spacing between representable values changes depending on magnitude), exponent alignment (before two floating-point numbers can be added, the hardware must shift one to match the other's scale), normalization and rounding (after every operation, the result must be adjusted to fit back into the standard floating-point format), and complex arithmetic units built to handle all of this transparently. Each floating point operation requires multiple internal steps just to *prepare* the numbers for arithmetic. Each value occupies more memory. Each byte moved costs energy. As we've seen, inference systems are dominated not by computation, but by *data movement and power*. Floating point inflates both.

The key insight here is subtle but important: *floating point precision is often unused precision during inference*. The model does not demand it. The hardware still pays for it.

1.3.2 What integers give up, and what they gain

An integer format defines a closed numerical world. INT8, for example, uses 8 bits, which yields exactly 256 distinct values, all evenly spaced. There is no concept of "very large" or "very small" beyond the chosen range. Every value costs the same amount of storage. Every operation behaves predictably. Integers do not try to be clever. They are explicit and bounded in what they

can represent.

This strictness shows up immediately in hardware. A floating point operation must align exponents, perform mantissa arithmetic, normalize results, and handle rounding and edge cases. An integer operation does none of this. It operates on fixed-width values, produces a result in a predictable number of cycles, and requires far less control logic. As a result, integer arithmetic typically consumes fewer cycles, integer units draw less power per operation, and more integer ALUs can fit in the same silicon area. This is not a micro-optimization. It is a structural advantage. On modern CPUs and GPUs, integer pipelines are shallower, denser, and often capable of issuing more operations per cycle. This leads to higher throughput and lower energy per token.

You might worry that this rigidity collapses accuracy. It does not, and the reason is the same one we covered in section 1.2: trained networks encode relative influence and direction, not exact values, and they absorb the small bounded perturbations of a fixed grid the way they already absorb training noise. What is new here is the *shape* of the perturbation: uniform across the grid's range, rather than dense near zero and sparse for large magnitudes. We will quantify that shift in chapter 2.

At the heart of quantization lies a single unavoidable tradeoff. You must decide how much *range* you need and how much *resolution* you can afford to lose. Floating point preserves both by paying a high system cost. Integers force you to choose. You cannot represent everything. You must decide what matters. This tension,range versus resolution, will appear in every quantization decision you make, whether you are choosing bit-widths, grouping schemes, or calibration strategies.

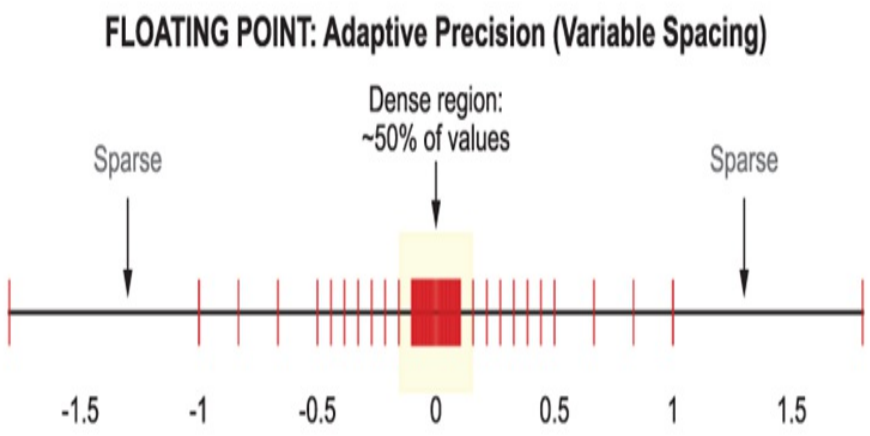
A useful way to think about this transition is through analogy.

Floating point is like a *zoom lens*: it constantly adjusts to keep objects in focus, no matter how far or close they are.

Integers are like a *fixed grid*: once you place it, everything must align to it. Quantization is the act of choosing where to place that grid. Once placed, the grid does not move. The numbers must fit.

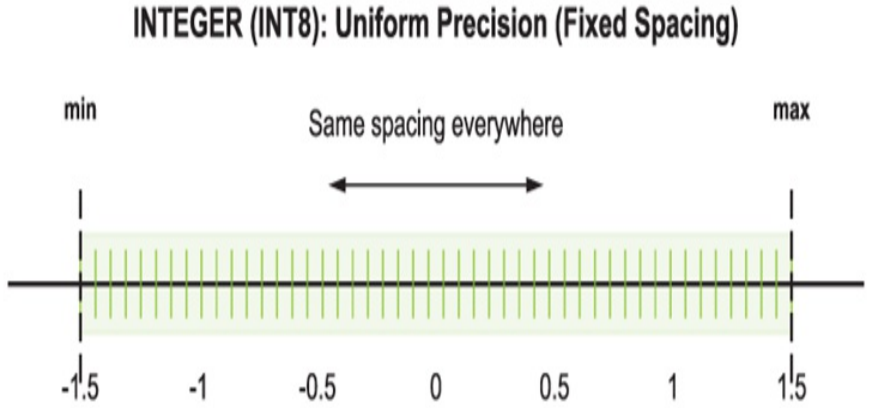
Figure 1.3 makes this distinction concrete by showing how floating-point adapts its spacing to where values cluster, while integer quantization commits to a single, fixed grid over a chosen range.

Figure 1.3 Floating point concentrates precision near zero (top), leaving large values sparsely represented. Integers use uniform spacing across your chosen range (bottom). Quantization is the act of deciding where to place that fixed grid.



QUANTIZATION

#A Choose range to cover
#B Divide into 256 equal steps
#C Map each value to nearest step



Key insight: You choose where to spend your 256 values.
Place the grid where your weights actually live.

The third mental anchor of this book

Floating point maximizes expressiveness. Integers maximize efficiency. Quantization is the art of deciding how much expressiveness you actually need.

We’re now ready to leave intuition behind and build quantization from first principles. The next chapter introduces the numerical machinery that makes this transition possible and the specific kinds of error it creates.

1.4 Summary

- Modern LLM inference is constrained by memory bandwidth and power consumption, not raw compute; a 7B parameter model moves roughly 15 GB through memory for every token generated, consuming nearly 2 joules of energy per token.
- Lowering precision from 16-bit floats to 8-bit or 4-bit integers cuts memory footprint, bandwidth, and energy in roughly the same proportion as the bit reduction; that direct, linear scaling is what makes quantization the practical lever rather than a theoretical one.
- Neural networks tolerate quantization because they encode directions and correlations rather than exact values; their inherent redundancy absorbs the bounded approximation error that lower precision introduces.
- The core tradeoff in quantization is range versus resolution: integers force you to choose a fixed grid where every number must fit, unlike floating point which auto-scales at the cost of hardware complexity.

2 Building Quantization from First Principles

This chapter covers

- Fixed-point number representation
- Affine quantization mapping
- Scale and zero-point parameters
- Quantization error analysis

Every number inside a neural network, every weight, every activation, every gradient, lives on the real number line. Training happens in floating-point, where that line stretches as far as the mathematics requires. Deployment is a different story. The devices that run inference at scale, including mobile phones, edge accelerators, and data-center INT8 cores, speak a cruder language: small integers packed into 8-bit or 4-bit containers. Quantization is the engineering discipline that translates between these two worlds, mapping continuous floating-point values onto a finite grid of discrete integer levels.

We previously established *why* this translation matters: smaller types mean less memory, faster arithmetic, and lower energy per inference.

Now, we build the translation itself. We start from the hardware up, looking at how a fixed-point machine actually stores and manipulates numbers, and work toward the affine mapping that modern frameworks use to compress neural network tensors into integers. Along the way, we derive the scale and zero-point parameters, analyze the two kinds of error that quantization introduces, and confront the engineering tradeoff between symmetric and asymmetric schemes.

If you are a machine learning engineer preparing to quantize a production model, this chapter gives you the mathematical vocabulary and geometric intuition to understand what your quantization toolkit is doing under the hood. If you are a systems engineer designing inference hardware, it explains

why the software stack makes the choices it does. And if you are simply curious about how a 32-bit floating-point weight becomes an 8-bit integer without destroying the model, this is where that story begins.

2.1 The fixed-point world

In the world of high-level software (Python, PyTorch), numbers are abstract entities. But in the fixed-point world of embedded DSPs and AI accelerators, numbers are physical states of hardware switches. To understand how a machine "thinks" about 1.5 or −0.5, we must first understand how it counts.

The fixed-point abstraction (Q-format)

Fixed-point is simply an agreement between you and the hardware: "We will store integers, but we will mathematically treat them as if they are divided by a constant Scale Factor (S)."

Equation 2.1

$$RealValue = Integer \times S$$

In binary hardware, S is always a power of two (2^{-n}). We use Q-format notation to track this. For example, Q3.4 (signed) means 3 integer bits and 4 fractional bits, giving a scale factor of $S = 2^{-4} = 1/16 = 0.0625$.

Walkthrough 1: the "1.5" example

Let's encode the number *1.5*. First, quantize: $1.5 \div S = 1.5 \times 16 = 24$. The hardware stores the integer *24*, which in binary is *0001 1000*. If we visualize the bits, we can see the "imaginary" binary point:

Figure 2.1 The Q3.4 fixed-point format divides 8 bits into a sign bit, 3 integer bits, and 4 fractional bits. Here, the bit pattern 0 001 1000 encodes +1.5 — the integer bits 001 contribute 1, and the leading fractional bit 1000 contributes $2^{-1} = 0.5$.

Walkthrough 2: the "1111 1111" example

Now, let's decode the confusing case: all ones. You read a memory address and find *1111 1111*. What number is this?

Q-format integers use two's complement representation. For an 8-bit signed integer, the range is −128 to +127, and the MSB (most significant bit) is the sign bit: 0 means positive, 1 means negative.

Let's write it with place values:

Table 2.1 Two's complement bit position values

Bit position	Value
b7 (sign)	−128
b6	+64
b5	+32
b4	+16
b3	+8
b2	+4
b1	+2
b0	+1

Now sum them: −128+64+32+16+8+4+2+1=−1

So 1111 1111 is the integer -1. Apply the scale by multiplying by the resolution (0.0625): −1×0.0625=−0.0625. So, 1111 1111 represents -0.0625. It is the negative number closest to zero.

Why 1111 1111 is −1: the odometer in your head

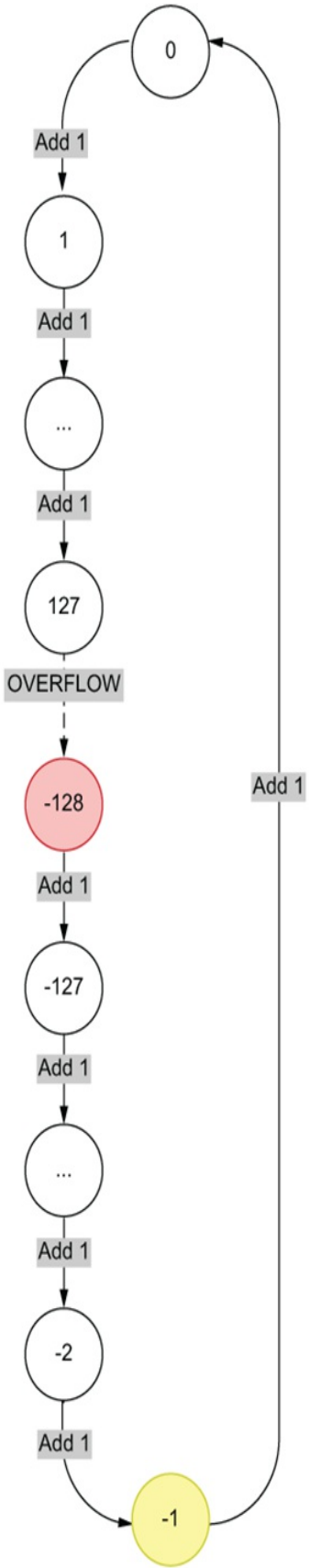
Table 2.1 told us *that* 1111 1111 sums to −1. It does not tell us *why* hardware ends up using that representation. The reason is mechanical, and a 3-digit car odometer is the easiest way to feel it.

The odometer reads 000. Drive backward one mile and the wheels physically roll the other way: 999. Add 1 to 999, the gears carry, the leading digit falls off the dial, and you're back at 000. In this system, 999 *behaves* like −1 because 999 + 1 = 0.

An 8-bit register is the same machine with 256 positions instead of 1,000. Subtract 1 from 0000 0000 and the hardware wraps backward to 1111 1111. That is the *mechanical* reason 1111 1111 is −1, and it is the reason the place-value trick in Table 2.1 works at all: the sign bit's −128 is exactly the wrap-around offset that makes the arithmetic come out right.

Visualize the integer ring in Figure 2.2:

Figure 2.2 The 8-bit signed integer ring. The 256 possible bit patterns wrap around a circle, with 0 at the top and −1 (1111 1111) one step below it. Moving clockwise increments the integer; moving counterclockwise decrements it.


0
Add 1
1
Add 1
...
Add 1
127
OVERFLOW
-128
Add 1
-127
Add 1
...
Add 1
-2
Add 1
-1
Add 1

This visualization highlights two critical boundaries. The *Overflow Cliff* is the point where incrementing the maximum positive value flips the sign bit: incrementing 127 by 1 produces the bit pattern 1000 0000, and the value warps from +127 straight to −128, the minimum negative number. The *-1 Bridge* is the pattern of "all ones" (1111 1111), which is always the integer −1, just one step below zero.

Python simulation: probing the boundaries

We can write a Python script to simulate this hardware. A real ALU (arithmetic logic unit, the hardware block that executes integer add, subtract, shift, and multiply) operates exclusively on integers. It has no concept of a decimal point. The "fixed point" is a convention the programmer maintains, not something the silicon knows about. Our simulation mirrors this: we store raw integers, perform integer arithmetic (shifts, adds, multiplies), and apply the scale factor only when displaying a human-readable result.

Two operators show up repeatedly in the listing below: `<<` is a left shift and `>>` is a right shift. For example, `1 << 7` yields `10000000`, the value 1 shifted 7 places to the left, which is 128.

Listing 2.1 FixedPoint class simulating hardware behavior

```
class FixedPoint:
    def __init__(self, value, int_bits=3, frac_bits=4):
        self.n = frac_bits
        self.scale = 1 << frac_bits
        total_bits = int_bits + frac_bits + 1
        self.min_int = -(1 << (total_bits - 1))
        self.max_int = (1 << (total_bits - 1)) - 1

        if isinstance(value, float):
            self.raw = int(round(value * self.scale))
        else:
            self.raw = value
        self.raw = max(self.min_int, min(self.max_int, self.raw))

    def __repr__(self):
        return f"{self.raw / self.scale} (Raw: {self.raw})"
```

```
    def __add__(self, other):
        res = self.raw + other.raw
        return FixedPoint(res, int_bits=3, frac_bits=self.n)

    def __mul__(self, other):
        wide_product = self.raw * other.raw
        rounding_bias = self.scale >> 1
        final_raw = (wide_product + rounding_bias) >> self.n
        return FixedPoint(final_raw, int_bits=3, frac_bits=self.n
```

With the class in place, we can exercise it the way an embedded engineer would: multiply a small value, push a large one past the saturation cliff, and add two mid-range values that overflow on accumulation. Listing 2.2 runs all three and prints the raw integer alongside the human-readable interpretation, so we can see exactly when the Q3.4 grid holds and when it gives up.

Listing 2.2 FixedPoint arithmetic demonstration

```
a = FixedPoint(1.5)
b = FixedPoint(0.5)
print(f"{a} * {b} = {a * b}")              #A

c = FixedPoint(10.0)                       #B
print(f"Input 10.0 -> Stored as {c}")

d = FixedPoint(5.0)
print(f"{d} + {d} = {d + d}")              #C
```

Output:

```
1.5 (Raw: 24) * 0.5 (Raw: 8) = 0.75 (Raw: 12)
Input 10.0 -> Stored as 7.9375 (Raw: 127)
5.0 (Raw: 80) + 5.0 (Raw: 80) = 7.9375 (Raw: 127)
```

What multiplication actually produces

When you first see the code for *__mul__*, there is a good chance you will ask "but what is rounding bias?"

When two fixed-point numbers are multiplied, *raw_a = a × 16* and *raw_b = b × 16*, so *wide_product = raw_a × raw_b*. Mathematically: *wide_product =*

(a × b) × 256. So after multiplication, the result effectively has twice as many fractional bits (8 instead of 4).

This happens because each input was already scaled by 2^4. When you multiply two scaled integers, their scaling factors multiply as well: $(a \times 2^4) \times (b \times 2^4) = (a \times b) \times 2^8$. Nothing special or additional is being done—the extra fractional bits appear automatically as a consequence of multiplying two fixed-point numbers.

To return to Q3.4 format, we must divide by 16: *final_raw ≈ wide_product / 16*. In hardware, this is done using a right shift (equivalent to dividing by 16): *final_raw = wide_product >> 4*.

Important

Right shift is not real division—it is integer division that drops the remainder.

What goes wrong without rounding

Right shift always rounds down. That means any fractional information below the cutoff is discarded, every multiplication introduces a small negative error, and the error is systematic, not random.

Why 0.3 × 0.3 looks wrong

At first glance, fixed-point multiplication can look "wrong" because the inputs themselves may not be representable exactly. In Q3.4, the only representable values are multiples of 1/16 = 0.0625: ... 0.1875, 0.2500, 0.3125, 0.3750 ... So 0.3 cannot be stored exactly. Encoding does this: *0.3 × 16 = 4.8, round(4.8) = 5*, stored value = *5 / 16 = 0.3125*. That is the quantization step: you snap 0.3 to the nearest grid point. Quantization error for the input: $0.3125 - 0.3 = +0.0125$.

The "true" float math is $0.3 \times 0.3 = 0.09$. But fixed-point is actually doing $0.3125 \times 0.3125 = 0.09765625$. So you should *expect* the fixed-point result to be closer to 0.0977 than to 0.09, because the inputs were already nudged upward.

Multiply the raw integers: *wide_product = 5 × 5 = 25*. After multiplication the scale is 256 (2^8), so 25 represents 25/256=0.09765625. Rescale without rounding: *final_raw = 25 >> 4 = 1, value = 1 / 16 = 0.0625*. The shift discards the remainder, introducing a second error on top of the input quantization. Error relative to the true float result: 0.0625−0.09=−0.0275. This downward pull happens systematically if you always truncate.

Why the rounding bias fixes systematic truncation drift

Before shifting, we add half the scale: *rounding_bias = 8, final_raw = (wide_product + 8) >> 4*. This implements *round(wide_product / 16)* instead of *floor(wide_product / 16)*.

Same example, with rounding bias: *(25 + 8) >> 4 = 33 >> 4 = 2, 2 / 16 = 0.125*. Error: 0.125−0.09=+0.0350.

Wait, but didn't this increase the error? Yes, locally, rounding bias can increase error. And that is not bad. Rounding does not minimize error per operation; it minimizes error accumulation. Think of truncation as *always braking* and rounding as *sometimes braking, sometimes accelerating.* Braking might feel "safer" locally, but after 1,000 steps, you're at zero speed.

Key takeaways from the simulation

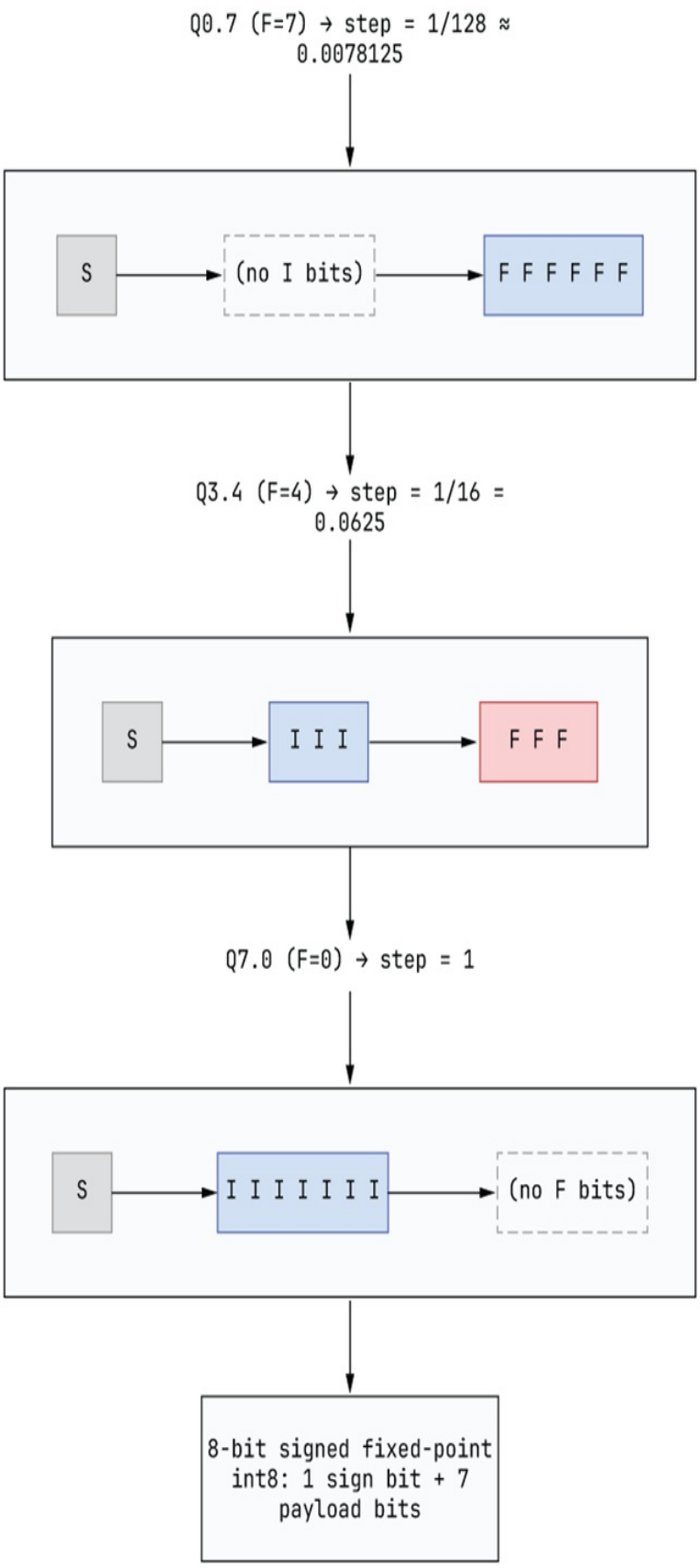
Max Positive (0111.1111) has interpretation 127 and value 127/16 = 7.9375. Max Negative (1000.0000) has interpretation -128 and value -128/16 = -8.0000. Note that the range is asymmetric: we can go down to -8.0, but only up to 7.9375; we cannot represent +8.0. The step size, the difference between any two bit patterns, is exactly 1/16 (0.0625). This is our quantization resolution.

The tradeoff: range vs. precision

The "tradeoff" mentioned here refers to the zero-sum game of allocating bits. With a fixed container (e.g., 8 bits), every bit we give to the integer part increases our range (how big the number can be) but decreases our precision

(how detailed the number can be).

Figure 2.3 The range-versus-precision tradeoff in Q-format. For a fixed 8-bit container, every bit allocated to the integer part doubles the representable range but halves the precision (step size). Moving the binary point left (more fractional bits) gives finer resolution at the cost of a narrower range; moving it right does the reverse. There is no free lunch: the total information capacity is fixed by the bit-width.


Q0.7 (F=7) → step = 1/128 ≈ 0.0078125
S
(no I bits)
F F F F F F
Q3.4 (F=4) → step = 1/16 = 0.0625
S
I I I
F F F
Q7.0 (F=0) → step = 1
S
I I I I I I I
(no F bits)
8-bit signed fixed-point int8: 1 sign bit + 7 payload bits

In deep learning quantization, selecting the correct Q-format is critical. If we choose Q0.7 for a neural network layer that produces values like 5.2, those values will hit the "Overflow Cliff" and be clamped to 0.99; the network will lose massive amounts of information (saturation). If we choose Q7.0 for a layer with tiny gradients like 0.001, they will all round to 0; the network will stop learning (vanishing gradients).

This necessitates calibration, the process of running data through the model to measure the actual dynamic range of activations, so we can pick the right spot for our binary point.

Why floating-point exists

Everything we have built so far lives entirely inside the fixed-point world: one grid, chosen once, applied everywhere. That works only if all values stay where we expected them to stay. Real numbers, especially inside neural networks, do not behave that politely.

Imagine you are forced to represent both small values (0.0008, 0.0012, 0.002) and large values (120.0, 256.0, 1024.0) with the same Q3.4 grid. If the grid is tuned for the large values, the small ones collapse to zero. If it is tuned for the small values, the large ones hit the overflow cliff. No bit allocation satisfies both. This is a fundamental geometric constraint, not a bug.

Floating-point solves exactly this problem: it lets the grid move. Instead of *value = integer × fixed_scale*, floating-point stores *value = mantissa × 2^exponent*, where the **mantissa** carries the significant digits of the number (the "1.6" in "1.6 × 2⁻⁴") and the **exponent** picks the binary order of magnitude. The exponent moves the ruler: small exponents produce fine tick marks for small numbers, large exponents produce coarse tick marks for large numbers.

Numerical intuition (a tiny "toy float")

To make this concrete without diving into IEEE-754 (the standard that defines how floating-point numbers are encoded in hardware), imagine a toy

floating-point format where the mantissa is stored as 1.x with 4 fractional bits and the exponent is an integer power of two. First, write each number in "binary scientific notation" (approximately): $0.01 \approx 1.28 \times 2^{-7}$, $0.10 \approx 1.60 \times 2^{-4}$, $1.00 = 1.00 \times 2^{0}$, $3.00 = 1.50 \times 2^{1}$.

Table 2.2 Quantize only the mantissa to the nearest multiple of 1/16

True Value	(Approx) mantissa × 2^exp	Quantized mantissa	Reconstructed value
0.01	1.28×2^{-7}	1.3125×2^{-7}	0.01025390625
0.10	1.60×2^{-4}	1.6250×2^{-4}	0.1015625
1.00	1.00×2^{0}	1.0000×2^{0}	1.0
3.00	1.50×2^{1}	1.5000×2^{1}	3.0

Notice what happened: 0.01 did not collapse to 0.0. The mantissa only had 4 fractional bits, the same "16 steps" idea as Q3.4, yet it preserved detail for small numbers. The step size in our toy float depends on the exponent: $\Delta \approx 2^{\text{exponent}} \times (1/16)$. Small numbers automatically get a finer ruler. That is the entire point of floating-point.

Not all 16-bit floats are the same

The exponent–mantissa split is a design choice, not a law of nature. Two 16-bit floating-point formats dominate deep learning, and they make opposite bets with their bit budget.

IEEE FP16 allocates 5 bits to the exponent and 10 to the mantissa. This gives fine precision—the mantissa grid has 1,024 steps—but a narrow dynamic range that tops out around 65,504. Values above that overflow to infinity, which is why FP16 mixed-precision training requires loss scaling to prevent gradient overflow.

BF16 (brain floating-point) flips the tradeoff: 8 exponent bits and only 7 mantissa bits. The mantissa grid has just 128 steps, 8× coarser than FP16, but the 8-bit exponent matches FP32's, giving a dynamic range up to $\sim 3.4 \times 10^{38}$.

Gradients that overflow in FP16 fit comfortably in BF16 without any scaling tricks. This is why BF16 became the default training format for LLMs: it is robust where FP16 is fragile.

For quantization, this distinction has a practical consequence. A model trained in BF16 has already learned to tolerate coarse mantissa precision—its weights and activations have been shaped by 128-step rounding throughout training. Quantizing such a model to INT8 (256 levels) is often gentler than quantizing an FP16-trained model, because the network never relied on the fine precision that INT8 removes.

The structural tradeoff

Fixed-point asks: *what is the one ruler we will use for everything?* Floating-point asks: *what ruler does this number need right now?*

Floating-point buys flexibility with hardware: every operation must align exponents, shift mantissas, renormalize, and round. Each step costs area, latency, and energy—exactly the costs Chapter 1 already counted. Fixed-point spends numerical flexibility to buy hardware efficiency.

Quantization exists because deep learning models tolerate a surprising amount of numerical rigidity—as long as we choose the grid intelligently. That is the problem affine quantization is designed to solve.

We now have everything we need: a concrete understanding of fixed-point arithmetic, a physical intuition for overflow and precision loss, and a clear reason why scale must be data-dependent. The next step is to formalize this intuition into a mapping that can be applied to any tensor, one that lets us choose both the step size and the offset of the integer grid. That mapping is the affine transformation, and deriving it is the subject of the next section.

2.2 Define the affine mapping with scale and zero point

The fixed-point grid we built in Section 2.1 was rigid by design: its scale was

locked to a power of two, and its center was pinned at zero. That rigidity made the hardware cheap, but it also meant the grid could not adapt to the data it was asked to represent. This section removes both constraints. We derive the affine quantization mapping, a linear transformation that lets us choose any scale and any offset, and show how it bridges the gap between the chaotic statistics of neural network tensors and the integer arithmetic of inference hardware.

The payoff is practical. By the end of this section you will be able to calculate the scale factor S and zero-point Z for any tensor, understand why exact representation of zero is non-negotiable, and see where the computational cost of the zero-point shows up in a matrix multiply. These are the building blocks that every quantization framework, including TensorFlow Lite, PyTorch, and ONNX Runtime, uses internally.

Why fixed-point breaks down

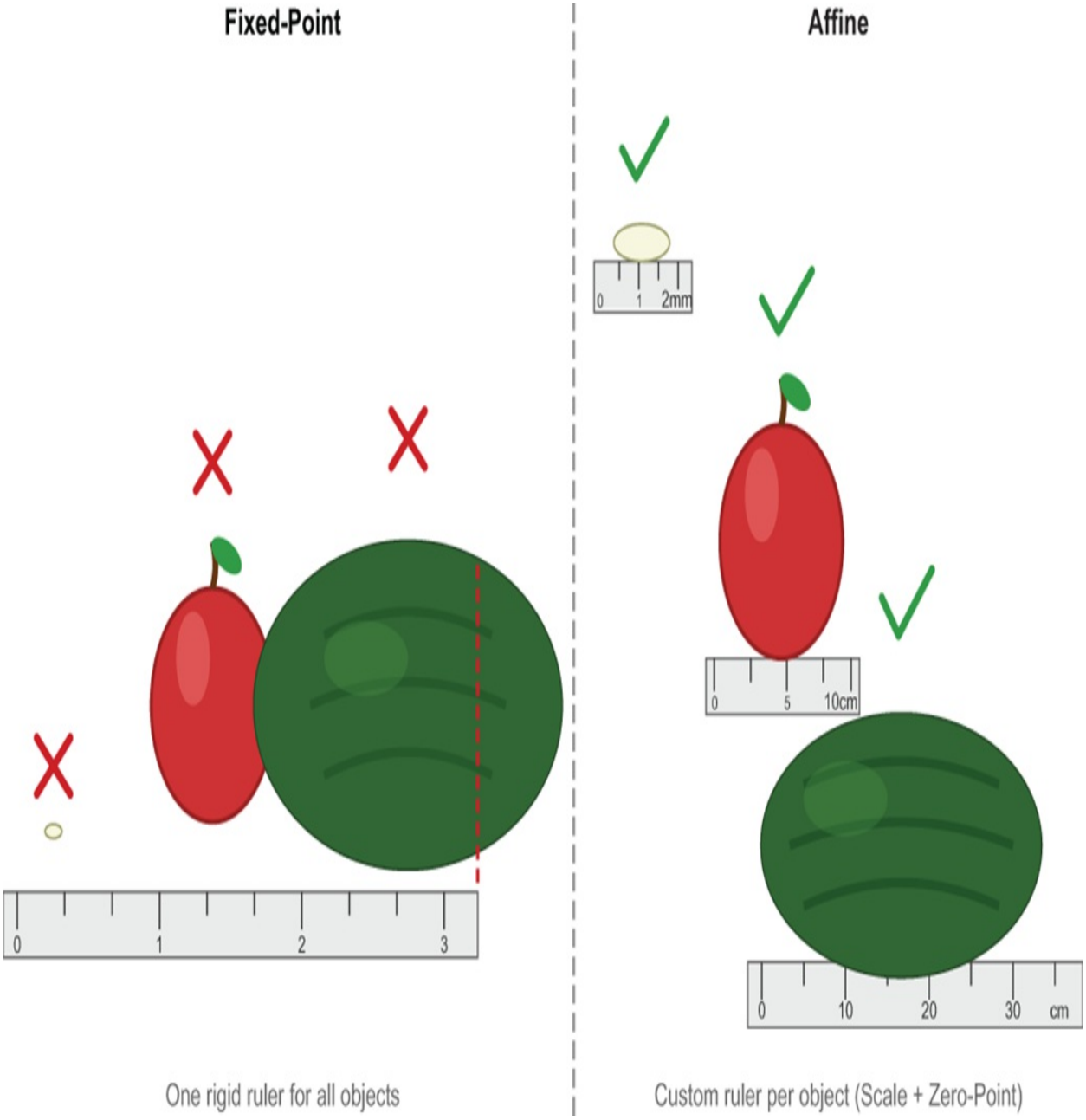
Neural networks produce tensor distributions of wild variety. A convolutional layer's weights might form a tight bell curve in [−0.05, +0.05], while the next layer's ReLU activations (the rectifier $f(x) = max(0, x)$, which zeros every negative input) span [0.0, +65.0]. No single Q-format can serve both: a wide-range format smashes the tiny weights to zero, while a high-precision format clips the large activations. Worse, ReLU outputs are strictly non-negative, yet a signed 8-bit integer reserves half its codes for negative values that never appear. This wastes an entire bit of representational capacity.

The ruler analogy

Imagine measuring a grain of rice, an apple, and a watermelon with a single standard ruler. The rice grain vanishes between tick marks; the watermelon extends past the edge and gets clipped. Only the apple, by luck, falls in a measurable range. Fixed-point quantization faces the same dilemma: one rigid grid cannot serve diverse distributions.

Figure 2.4 The ruler problem. Fixed-point quantization (left) forces a single rigid ruler on all measurements: a rice grain vanishes between coarse tick marks, while a watermelon extends beyond the ruler's edge and gets clipped. Affine quantization (right) manufactures a custom ruler for each object, with tick spacing (scale) and starting position (zero-point) tailored to the data's

actual range.



We decouple the storage format from the mathematical interpretation. Instead of a fixed power-of-two scale, we manufacture a custom ruler for every tensor, choosing the distance between tick marks (scale) and where the ruler starts (zero-point). Mathematically, this is an affine transformation: a linear scaling plus a translation. It maps an arbitrary floating-point range [α, β] onto the full integer range with maximal fidelity. This flexibility is the foundation

of all modern integer-only inference engines.

The geometry of the affine transformation

To understand affine quantization, one must move beyond the bit-level view of Section 2.1 and adopt a geometric perspective. Imagine the real number line as a continuous, infinitely stretchable elastic band. Imagine the integer number line (e.g., the 8-bit range [−128,127]) as a rigid, fixed grid of discrete slots.

Quantization is the process of mapping the elastic band onto the rigid grid.

In the fixed-point world, we were allowed to stretch or shrink the band only by factors of two (2,4,8,0.5,0.25...). Crucially, we were forced to pin the center of the band (real zero) to the center of the grid (integer zero). We could zoom in or out, but we could not look sideways.

In the affine world, we grant ourselves two new freedoms. *Arbitrary Scaling (S)*: We can stretch or shrink the real number line by any positive real factor. This allows us to perfectly match the width of our data distribution to the width of the integer container. If our data spans a range of 3.5 units and we have 255 integer slots, we set the step size to exactly 3.5/255. *Arbitrary Shifting (ZZ Z):* We can slide the real number line left or right relative to the integer grid. This allows us to align the active region of our data with the available integers. If our data is centered at +100.0, we slide the grid so that the integers focus on that region, rather than wasting bits representing 0.0.

The fundamental equations

Let us formalize this relationship. We define r as the real-world value (typically float32), representing the actual weight, activation, or bias in the neural network. We define q as the quantized integer value (typically int8 or uint8), the value actually stored in memory and processed by the ALU. We define S as the scale factor (a positive float32), representing the step size or resolution of the quantization. And we define Z as the zero-point (an integer, same type as q), representing the integer value that corresponds to the real value 0.0.

The relationship between the real value and the quantized integer is defined by the dequantization equation (or reconstruction equation). This is the formula the hardware uses to "understand" what an integer means:

Equation 2.2

$$r = S \cdot (q - Z)$$

This equation is an affine map of the form $y=mx+c$, where $m=S$ and the intercept is determined by $-S \cdot Z$. It tells us that to recover the real value, we first “untare” the integer by subtracting the zero-point offset, and then scale the result by the step size S.

To go the other direction, converting a floating-point tensor into integers, we invert this equation to get the quantization equation:

Equation 2.3

$$q = round\left(\frac{r}{S} + Z\right)$$

In practice, because the integer q has a limited bit-width (e.g., 8 bits), we must also strictly enforce boundaries to prevent overflow. We apply a clamping (or clipping) function:

Equation 2.4

$$q = clip\left(round\left(\frac{r}{S} + Z\right), q_{min}, q_{max}\right)$$

where $round(\cdot)$ rounds to the nearest integer (the specific rounding tie-breaking rule is a subtle source of drift between frameworks), $clip(v,min,max)=min(max(v,min),max)$, and q_{min},q_{max} are the limits of the integer type (e.g., -128 and 127 for signed int8).

The system of linear equations

A common question arises: "How do we calculate S and Z for a given

tensor?" This is a deterministic calculation derived from the constraints of the mapping.

We start by identifying the boundaries of our real data distribution. Let r_{min} and r_{max} be the minimum and maximum values in the floating-point tensor. We want to map this range $[r_{min}, r_{max}]$ to the full range of the integer type $[q_{min}, q_{max}]$.

This gives us a system of two linear equations with two unknowns (S and Z).

Constraint 1: The real minimum must map to the integer minimum:
$r_{min} = S(q_{min} - Z)$

Constraint 2: The real maximum must map to the integer maximum:
$r_{max} = S(q_{max} - Z)$

To solve for the scale factor (S), we subtract constraint 1 from constraint 2 above:

$$r_{max} - r_{min} = S(q_{max} - q_{min})$$

Equation 2.5

$$S = \frac{r_{max} - r_{min}}{q_{max} - q_{min}}$$

This result is physically intuitive: the scale factor S is simply the ratio of the real range to the integer range. It represents the "value per bit" or the physical magnitude of a single integer step.

To solve for the zero-point (Z), we rearrange constraint 1:

Equation 2.6

$$Z = q_{min} - \frac{r_{min}}{S}$$

In a perfect mathematical world, both derivations would yield the same

result. But there is a catch: Z must be an integer. It represents a coordinate on the discrete grid. The value calculated by $q_{min} - r_{min}/S$ will almost certainly be a floating-point number. Therefore, we must round it:

Equation 2.7

$$Z = round\left(q_{min} - \frac{r_{min}}{S}\right)$$

This rounding step is critical. It implies that we cannot perfectly map the arbitrary range $[r_{min}, r_{max}]$ to $[q_{min}, q_{max}]$. By forcing Z to be an integer, we introduce a slight shift, effectively "nudging" the real range that our quantized tensor represents. This "nudge" is the first source of quantization error we encounter, distinct from the rounding error of individual values.

The scale factor S: resolution and granularity

The scale factor S is the parameter that allows the quantization scheme to "zoom in" or "zoom out" on the data. Unlike the fixed-point world, where resolution was locked to powers of two (e.g., $0.5, 0.25, 0.125, \ldots$), the affine scale factor can be any positive real number.

For example, if a weight tensor ranges from −0.123 to +0.123, and we use int8 (range 255), the scale factor would be

$$S = \frac{0.123 - (-0.123)}{255} \approx 0.0009647$$

This specific value becomes the "quantum" of our tensor—the smallest detectable difference between two values.

The granularity of scale

An important distinction in affine quantization is the granularity of S and Z. At what level do we define these parameters?

Per-Tensor quantization

Per-tensor quantization calculates a single pair of (*S*,*Z*) for the entire tensor. This is the simplest approach and the most memory efficient, as we only store two extra FP32 values per tensor. But it is highly susceptible to outliers. If a single neuron in a layer outputs a value of 100.0 while the rest are in the range [0,1.0], the per-tensor calculation will expand the range to [0,100.0]. The step size *S* will become huge (*100/255≈0.4*), completely obliterating the precision of the vast majority of the data in the [0,1.0] range.

Per-Channel quantization

Per-channel quantization calculates a unique (S,Z) pair for each channel (typically the output channel for weights). This effectively gives each filter in a convolutional neural network (CNN) its own custom ruler. This isolates the effect of outliers; a filter with large weights gets a large *S*, while a filter with small weights gets a small *S*. This significantly improves accuracy but requires the hardware to support switching scale factors frequently during the dot product accumulation.

Between these two extremes sits group-wise quantization: each channel is partitioned into smaller blocks (typically 32, 64, or 128 elements), and every block gets its own scale. This middle ground limits outlier damage where per-channel still costs too much, an arrangement that becomes load-bearing for 4-bit LLM weights, where neither per-tensor nor per-channel granularity holds up alone. Across the full spectrum, the choice trades model accuracy against hardware complexity.

The zero-point Z: anchoring the void

The zero-point Z is arguably the most significant innovation of affine mapping over traditional fixed-point representation. It is conceptually simple, an integer offset, but its implications for neural network accuracy and sparsity are profound.

The "Tare" analogy

To understand Z, consider the operation of a digital kitchen scale. When you

place a bowl on the scale, it registers a weight. You do not care about the weight of the bowl; you care about the flour you are about to add. So, you press the Tare button. The display resets to zero. Internally, the load cell is still supporting the weight of the bowl, but the digital logic subtracts that offset before displaying the result.

In affine quantization, the integer (q) is the raw measurement of the load cell, the zero-point (Z) is the tare value, the weight of the bowl, and the real value (r) is the flour, the value displayed to the user.

We "tare" the integer grid so that the mathematical zero aligns with a specific integer coordinate. If our data ranges from 2.0 to 5.0, we don't want the integer 0 to represent mathematical 0.0, because mathematical 0.0 is irrelevant to our data. We want the integer 0 (or -128) to represent 2.0, effectively shifting our window of focus to where the information actually lives.

The absolute necessity of exact zero

One might ask: "Why must Z be an integer? Why not allow a fractional offset for better precision?" Or, "Why do we force the range to include zero?"

The answer lies in the specific operations of deep learning: zero-padding and sparsity.

Zero-padding in convolutions

Convolutional layers frequently use "zero-padding" to maintain spatial dimensions (e.g., `padding='same'`). This involves surrounding the input tensor with values that are mathematically 0.0. In the quantized world, we perform this padding on the integer matrix. We must pad with some integer value. If real 0.0 does not map to an exact integer, we have a problem.

Suppose our quantization maps real 0.0 to the fractional value $Z=12.4$. Since we can only store integers, we must round Z to 12. But integer 12 corresponds to a real value slightly different from zero (e.g., 0.05). If we pad our image with the integer 12, we are effectively surrounding our image with

a border of value 0.05. This "gray border" creates a massive artificial edge signal at the boundary of the image. The convolution filters, which are edge detectors, will fire wildly at these borders, introducing significant noise and degrading accuracy.

By forcing Z to be an integer and calculating S such that 0.0 is exactly representable, we ensure that we can pad the integer tensor with the value Z and mathematically achieve "real zero" padding.

ReLU and Sparsity

The ReLU activation function ($y=max(0,x)$) sets all negative values to exactly 0.0. In sparse networks, these zeros can be skipped during computation to save energy (zero-skipping). If 0.0 is not exactly representable, ReLU outputs become small nonzero noise (e.g., 0.001). The sparsity is destroyed. The hardware cannot skip these values, and the "dead neurons" effectively leak signal, altering the network dynamics.

Thus, a cardinal rule of affine quantization is the exact zero constraint: *The real value 0.0 must map to an integer Z without any quantization error.*

This constraint often forces us to adjust our min/max ranges. If a tensor has a range of [1.0,5.0], we must artificially extend the range to [0.0,5.0] during calibration solely to ensure that 0.0 is included and can be anchored to an integer.

The arithmetic of affine quantization: the cost of Z

We have established that the affine mapping $r=S(q-Z)$ is geometrically superior to fixed-point because it fits the data perfectly. But there is no free lunch. The cost of this flexibility is paid in arithmetic complexity.

In the fixed-point world, a dot product between a weight w and an input x is simple: $y=\sum(w\ x)\approx S_{out}$. It is a straight integer dot product scaled at the end.

In the affine world, we substitute the affine definitions into the dot product. Substituting $r_w=S_w(q_w-Z_w)$:

Equation 2.8

$$y = S_w S_x \Sigma((q_w - Z_w)(q_x - Z_x))$$

Now we must expand the term inside the summation. Recall the algebraic expansion *(a−b)(c−d)=ac−ad−bc+bd*:

$$y = S_w S_x \Sigma(q_w q_x - q_w Z_x - Z_w q_x + Z_w Z_x)$$

Distributing the summation across these terms gives us four distinct components that the hardware must compute:

Table 2.3 Affine dot product expansion terms

Term	Formula	Description	Computational Characteristic
1. Core Product	$\sum(q_w q_x)$	Standard integer dot product	Expensive. Main O(N) MAC operation.
2. Weight Sum	$-Z_x \sum q_w$	Weights scaled by input zero-point	Pre-computable. $\sum q_w$ is constant for trained model.
3. Input Sum	$-Z_w \sum q_x$	Inputs scaled by weight zero-point	Runtime Cost. $\sum q_x$ depends on input.
4. Offset Constant	$+N \cdot Z_w \cdot Z_x$	Constant offset	Pre-computable. Can fold into bias.

The cross-term penalty

The expansion reveals that affine quantization introduces significant overhead compared to fixed-point: four terms where fixed-point needed one. Symmetric quantization eliminates two of them. Setting $Z_w = 0$ zeroes out Term 3 and Term 4 entirely, as the derivation of equation 2.9 will show. Specifically, Term 3 is the expensive one. It requires the inference engine to perform a reduction (summation) over the input activations for every dot

product.

This arithmetic complexity leads to a common optimization in the industry. Hardware designers often prefer symmetric quantization for weights (forcing $Z_w=0$).

If $Z_w=0$, then Term 3 ($-Z_w\sum q_x$) and Term 4 ($+\sum Z_w Z_x$) vanish entirely. The equation simplifies to:

Equation 2.9

$$y = S_w S_x(\Sigma q_w q_x - Z_x \Sigma q_w)$$

This eliminates the runtime overhead of summing the inputs, retaining the efficiency of fixed-point while still allowing the activations to be asymmetric (keeping $Z_x \neq 0$ to handle ReLU).

Algorithmic implementation and simulation

To solidify the transition from mathematical theory to practical application, it is instructive to simulate the affine quantization process in code. The following Python implementation demonstrates the derivation of S and Z, the application of the "Nudge" to ensure zero inclusion, and the rounding/clipping behavior that characterizes the integer grid.

Listing 2.3 AffineQuantizer class

```
import numpy as np

class AffineQuantizer:
    def __init__(self, bit_width=8, signed=False):
        if signed:
            self.q_min = -(1 << (bit_width - 1))              #A
            self.q_max = (1 << (bit_width - 1)) - 1
        else:
            self.q_min = 0                                     #
            self.q_max = (1 << bit_width) - 1
        self.scale = None
        self.zero_point = None
```

```
    def calibrate(self, tensor):
        r_min, r_max = tensor.min(), tensor.max()
        r_min = min(r_min, 0.0)                                    #
        r_max = max(r_max, 0.0)                                    #

        real_range = r_max - r_min
        if real_range == 0:
            self.scale, self.zero_point = 1.0, 0
            return

        self.scale = real_range / (self.q_max - self.q_min)    #D
        initial_z = self.q_min - (r_min / self.scale)
        self.zero_point = int(np.round(initial_z))                 #
        self.zero_point = max(self.q_min, min(self.q_max, self.ze

    def quantize(self, r):
        q_scaled = (r / self.scale) + self.zero_point
        q_rounded = np.round(q_scaled)
        return np.clip(q_rounded, self.q_min, self.q_max)          #

    def dequantize(self, q):
        return self.scale * (q.astype(np.float32) - self.zero_poi
```

Listing 2.4 below puts the mechanism to work on a small ReLU-style tensor, the canonical asymmetric distribution: a few hard zeros and a long positive tail. We feed it to the quantizer, inspect the derived scale and zero-point, and check that round-tripping the values through `quantize` and `dequantize` gives us back the originals to within the bounded ±S/2 error we expect.

Listing 2.4 Calibrating and quantizing ReLU data with AffineQuantizer

```
data_relu = np.array([0.0, 0.0, 0.25, 1.5, 6.0], dtype=np.float32
quant = AffineQuantizer(bit_width=8, signed=False)

quant.calibrate(data_relu)                                         #
q_integers = quant.quantize(data_relu)                             #
data_recon = quant.dequantize(q_integers)                          #

print(f"Scale: {quant.scale:.6f}, Zero-Point: {quant.zero_point}"
print(f"Original:      {data_relu}")
print(f"Quantized:     {q_integers}")
print(f"Reconstructed: {data_recon}")
print(f"Error:         {data_relu - data_recon}")
```

Output:

```
Scale: 0.023529, Zero-Point: 0
Original:      [0.   0.   0.25 1.5  6.  ]
Quantized:     [  0   0  11  64 255]
Reconstructed: [0.         0.         0.25882354 1.5058824  6.
Error:         [ 0.         0.        -0.00882354 -0.00588238
```

Analysis of the simulation

The simulation highlights several theoretical concepts in practice. The zero-point nudge: the line $r_{min} = min(r_{min}, 0.0)$ implements the exact zero constraint. Without it, a tensor with values [1.0, 6.0] would map 1.0 to integer 0, and real 0.0 would map to a negative integer, which is impossible in uint8. Saturation: the *clip* function handles values

outside the calibrated range; an activation of 7.0 (when calibrated max was 6.0) clamps to 255, introducing unbounded error. Quantization noise: the difference between original and reconstructed is bounded by $\pm S/2$. Notice that for 0.0, the error is exactly 0.0—our zero-point logic works correctly.

Summary and forward outlook

Section 2.2 has guided us through the critical transition from the rigid hardware constraints of the past to the flexible data requirements of the present. We have replaced the "Power-of-Two" scalar with the affine map:

$$r = S \cdot (q - Z)$$

This mapping provides us with two powerful tools. The scale factor (S) is a custom resolution setting that allows us to stretch the integer grid to cover any dynamic range, from tiny weight gradients to massive activation spikes. The zero-point (Z) is a custom offset that allows us to align the integer grid with the data's center of mass, handling asymmetric distributions like ReLU and preserving the semantic integrity of zero padding and sparsity.

But we have also uncovered the price of this flexibility. The introduction of Z

complicates the fundamental arithmetic of the neural network, turning a clean dot product into a polynomial expansion with cross-terms ($Z_w \sum q_x$). This creates a tension between the mathematical optimality of affine quantization (it fits the data best) and the computational efficiency of symmetric quantization (it calculates faster).

Furthermore, our derivation of *S* and *Z* relied on a dangerous assumption: that we know the min (r_{min}) and max (r_{max}) of the data. For weights, this is easy; we can scan the file. But for activations, which change with every input image, how do we determine these bounds? Do we use the absolute maximum of a single batch? Do we use a moving average? What if a single outlier value of 1000.0 stretches our scale factor so thin that the rest of the signal vanishes?

These questions define the problem of calibration, which is the art of choosing the optimal r_{min} and r_{max} explored in the next section.

2.3 Characterize granular and overload errors

Quantization is lossy; that much is obvious. But not all losses are equal, and understanding *how* quantization distorts a signal is the key to controlling the damage. Two distinct failure modes govern every quantized tensor, and every calibration decision you will ever make reduces to a tradeoff between them. The goal here is to give you a felt sense of both, grounded in code and visuals rather than lengthy derivations. Everything reduces to a single picture in your head:

You place a fixed grid over a continuous signal.

Two bad things can now happen. Values land between grid points → Granular error. Values fall outside the grid → Overload error.

Granular error: when the grid is too coarse.

For uniform quantization with step size S, granular error is bounded. Maximum error magnitude is:

$$|error| \leq S/2$$

To see what that bound feels like, Listing 2.5 quantizes a small batch of values that all live well inside the grid's range. There is no clipping. The only thing that can go wrong is rounding. The step size S = 0.25 is deliberately coarser than the data spread (~±0.12), which forces every value to snap to the nearest grid point and reveals the bounded error in its purest form.

Listing 2.5 Granular error in practice

```
x = np.array([-0.12, -0.08, -0.03, 0.02, 0.07, 0.11])
S = 0.25                                                    #A
q = np.round(x / S) * S                                     #B

print("Real:     ", x)
print("Quantized:", q)
print("Error:    ", x - q)
```

Output:

```
Real:       [-0.12 -0.08 -0.03  0.02  0.07  0.11]
Quantized:  [-0.    -0.    -0.    0.    0.    0.  ]
Error:      [-0.12 -0.08 -0.03  0.02  0.07  0.11]
```

Listing 2.5 shows the bounded case. Every value sat inside the grid, so the worst rounding error stayed inside S/2 — predictable, recoverable, the kind of noise a downstream layer can absorb. Overload error is the unbounded twin: the grid runs out, and any value beyond the boundary gets clipped to that boundary. The further outside the data sits, the larger the error, with no ceiling.

Overload error injects large, structured distortion, creates sharp nonlinearities, and propagates strongly through layers. Consider a concrete case: a transformer's attention logits occasionally spike to 60.0, but your calibrated range covers [−20, 20]. Every spike is clamped to 20.0, a 3× reduction. After the softmax, those clipped logits produce attention weights that are dramatically flatter than intended, causing the model to attend almost uniformly instead of focusing on the correct token. A single clipped channel can corrupt an entire attention head.

The core tradeoff

You now know everything you need to understand calibration.

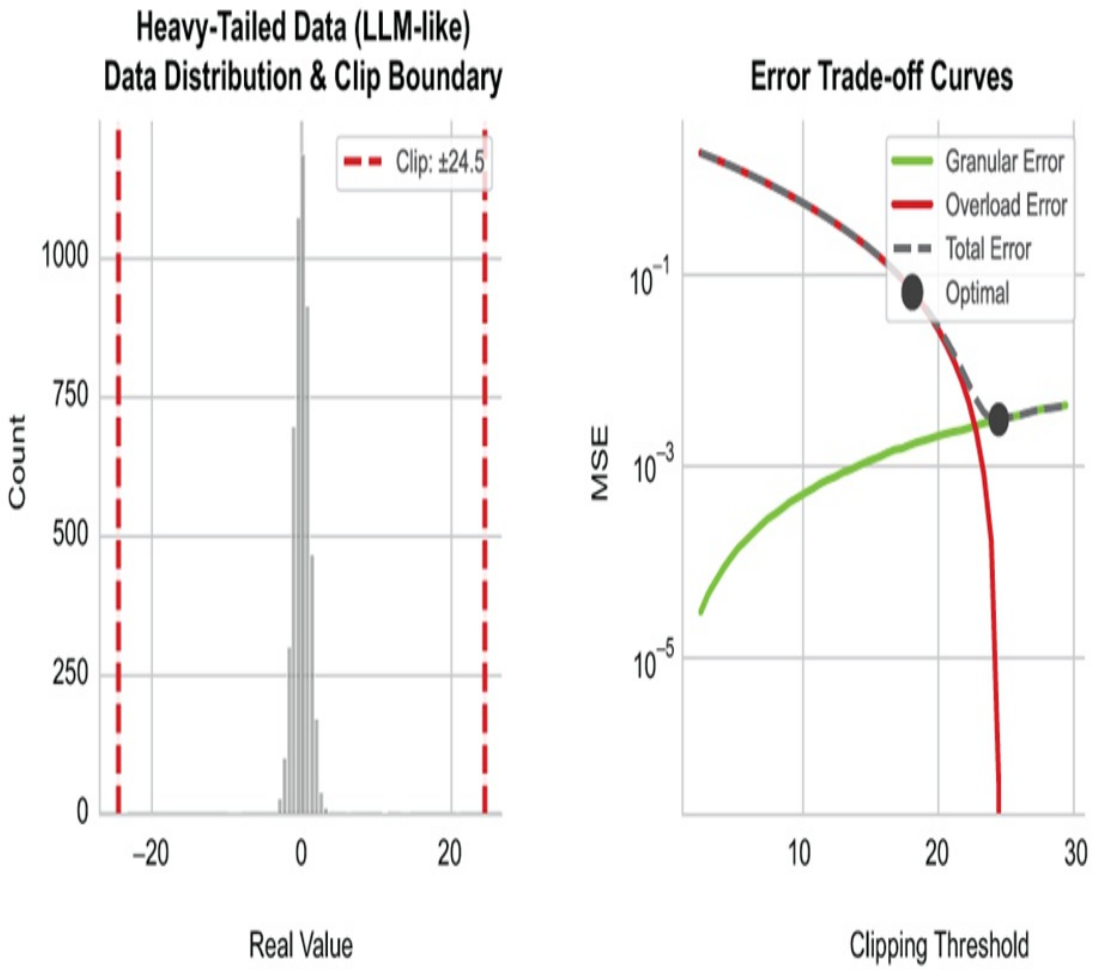
There are two knobs:

Knob 1: Make the grid wider. Fewer values clip, so overload error decreases. But step size increases, so granular error increases.

Knob 2: Make the grid tighter. Step size shrinks, so granular error decreases. But more values clip, so overload error increases.

Figure 2.5 makes this tradeoff visible. The two knobs correspond to a single slider: the clipping boundary that defines where the grid ends.

Figure 2.5 The granular-versus-overload error tradeoff. Widening the quantization grid (increasing the clipping range) reduces overload error—fewer values are clipped—but coarsens the step size, increasing granular error for values inside the range. Tightening the grid does the reverse. The optimal clipping boundary minimizes total distortion and depends on the shape of the data distribution: heavy-tailed distributions push the boundary outward; tightly concentrated distributions pull it inward.



In practice, the optimal boundary is rarely at the data's absolute min/max. Percentile-based calibration (e.g., clipping at the 99.99th percentile) deliberately accepts a small amount of overload error in the tails in exchange for a tighter grid and lower granular error across the bulk of the distribution. This is the principle behind calibration methods like entropy calibration (TensorRT) and mean-squared-error minimization (PyTorch)—each optimizes a different proxy for total distortion.

NOTE

You cannot minimize granular and overload error at the same time—every clipping boundary trades one for the other. Quantization is choosing which mistake you prefer.

Why outliers break everything (LLMs in particular)

Modern LLMs introduced a pathological case: 99.9% of values are small, 0.1% are massive. Example: most activations cluster in [−1,1], but one channel spikes to ~100.

If you protect the outlier, scale explodes and granular error destroys the majority. If you ignore the outlier, one channel clips hard and model behavior breaks.

This is why naive INT8 fails on transformers. And this is why later chapters introduce per-channel scaling, group-wise quantization, and outlier-aware methods.

A simple visualization trick

One quick way to predict whether a tensor will suffer more from granular or overload error is to plot the distribution against the proposed clipping boundaries before you commit to a scheme. Listing 2.6 provides the minimal recipe: histogram the activations and draw vertical lines at the symmetric clipping range ±α. Anything inside the lines pays granular error; anything outside pays overload error.

Listing 2.6 Visualizing quantization risk

```
plt.hist(activations, bins=100)
plt.axvline(-alpha, color='r')                    #A
plt.axvline(alpha, color='r')                     #A
plt.show()
```

Then ask: Is the center dense? → granular risk. Are the tails heavy? → overload risk. This single plot explains 80% of quantization failures.

Calibration is balancing these two. You cannot eliminate error. You can only decide where it shows up. That decision leads directly to the next question: Should the grid be symmetric—or should it move to follow the data?

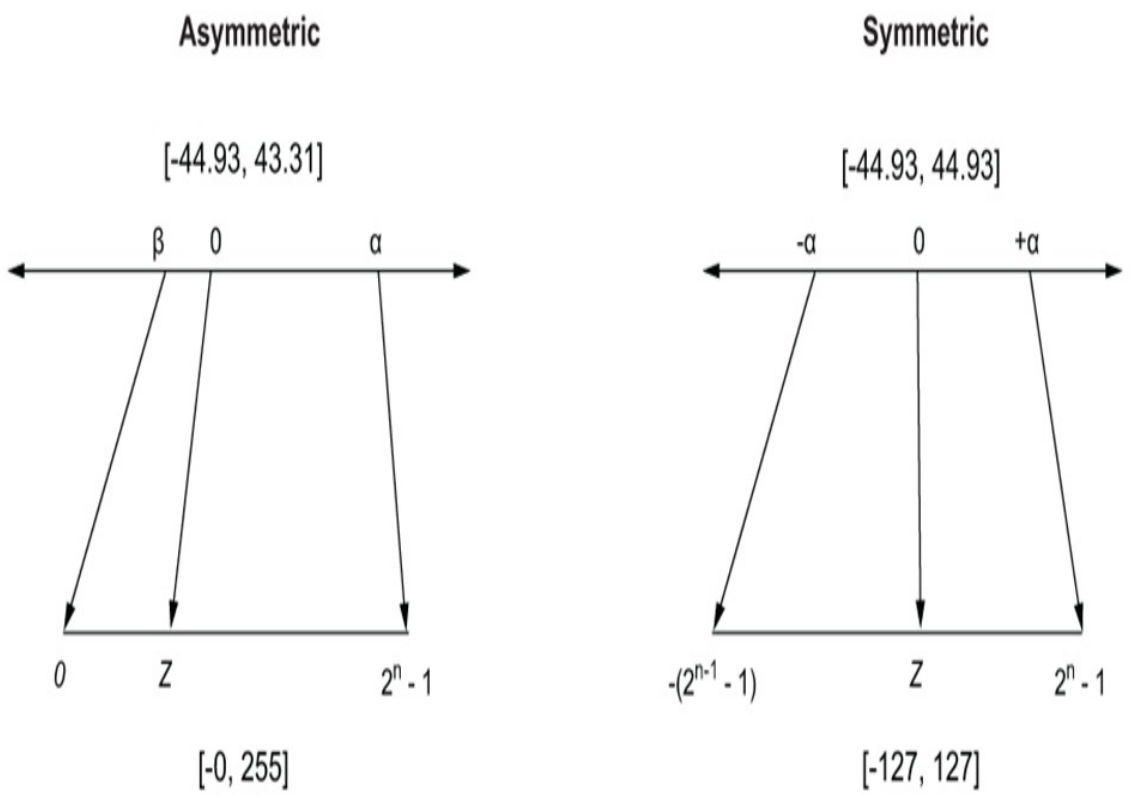
2.4 Compare symmetric and asymmetric choices

With the error landscape mapped out, the next design decision is the shape of the grid itself. Should it be centered on zero, or should it slide to follow the data? This is the symmetric-versus-asymmetric choice, and the Hybrid Consensus adopted by TensorFlow Lite, PyTorch, and ONNX Runtime exists because neither scheme wins everywhere.

Section 2.2 already gave us the mechanism. The affine dot product produces four terms; symmetric quantization ($Z = 0$) eliminates two of them and recovers the efficiency of fixed-point, while asymmetric keeps all four and pays the runtime cost to gain a perfect fit. Which form to use is not a hyperparameter to tune—it is dictated by the shape of the distribution being quantized. The rest of this section reads each distribution in turn and shows why the Hybrid Consensus falls out naturally.

Figure 2.6 makes that distinction visual. The asymmetric grid on the left fits itself to the data; the symmetric grid on the right keeps zero pinned at the center. The cost and benefit of each choice will fall out of the bell-curve and ReLU discussions that follow.

Figure 2.6 Symmetric versus asymmetric quantization. Asymmetric quantization (left) shifts the grid to cover only the region the data actually occupies, reclaiming bits that would otherwise encode empty space. The zero-point Z records the shift, and must be accounted for during inference arithmetic. Symmetric quantization (right) pins the grid at zero and mirrors the range, making it ideal for centered distributions like weight tensors but wasteful for one-sided data.



We have already observed that symmetric quantization is far more efficient than asymmetric quantization. Why do we struggle with asymmetric quantization if symmetric is so much faster? The answer lies in the probability density functions (PDFs) of the tensors we are trying to compress.

Weight distributions: the bell curve

Neural network weights are not random numbers; they are the result of stochastic gradient descent (SGD) with regularization (like L2 weight decay). Consequently, weight tensors almost universally exhibit a bell-shaped distribution (Gaussian or Laplacian) centered at zero. The mean $\mu \approx 0$, the distribution is symmetric such that $P(w) \approx P(-w)$, and values decay rapidly away from zero in the tails.

For such a distribution, Symmetric Quantization is statistically ideal. If a weight tensor spans [−0.5,0.5], a symmetric grid [−0.5,0.5] captures the data perfectly. Even if there is a slight shift (e.g., [−0.48,0.52]), enforcing symmetry to [−0.52,0.52] results in negligible wasted space.

Therefore, for weights, the "cost" of symmetric quantization (in terms of accuracy loss) is near zero. Combined with the "gain" of eliminating the runtime input sum ($Z_w=0$), Symmetric Quantization for Weights is the undisputed industry standard.

Activations behave differently. They are the output of non-linear functions like ReLU or GeLU.

ReLU (*f(x)=max(0,x)*) produces output distributions that are strictly non-negative $[0,x_{max}]$. It is a "half-distribution" with a massive spike at exactly 0 (sparsity). GeLU and Swish are mostly positive, but with a small negative dip (e.g., down to -0.17) before returning to 0.

This means using symmetric quantization will ensure half the range is wasted, as negative values never show up. This statistical reality drives the need for asymmetric quantization on the activation side. It is the only way to reclaim the bit-depth lost to the ReLU function.

The "Hybrid" consensus: symmetric weights, asymmetric activations

The collision of these hardware constraints and statistical realities has forged a pragmatic consensus in the deep learning community, adopted by frameworks like TensorFlow Lite, PyTorch Quantization, and ONNX Runtime. This configuration is known as the Hybrid Scheme: "Symmetric Weights, Asymmetric Activations."

This setup leverages the best of both worlds. *Computational Efficiency:* By forcing weights to be symmetric ($Z_w=0$), we eliminate Term 3 ($Z_w\sum q_x$) from the dot product expansion (Section 2.2). This removes the expensive runtime reduction sum over the input tensor, significantly speeding up inference. *Representational Density:* By allowing activations to be asymmetric ($Z_x\neq0$, or effectively uint8 with offset), we fully utilize the bit-width for skewed distributions like ReLU, avoiding the "7-bit trap" of symmetric signed quantization.

Table 2.4 lines up the three options side by side. Read it as a decision aid: the column you want for any given tensor depends on whether the data is

centered (favoring the symmetric column), one-sided or shifted (favoring asymmetric), or in practice, a mix of both inside the same layer, which is where the Hybrid Consensus column does its work.

Table 2.4 Quantization scheme comparison

Feature	Symmetric	Asymmetric	Hybrid Consensus
Zero Point (Z)	Fixed at 0	Calculated from min/max	$Z_w=0$, Z_x calculated
Grid Constraint	Centralized ($[-\alpha,\alpha]$)	Flexible ($[\alpha,\beta]$	Weights central, activations flexible
Compute Overhead	Lowest. Raw dot product.	Highest. Requires $\sum q_x$ and $\sum q_w$.	Low. Removes $\sum q_x$ overhead.
ReLU Efficiency	Poor. Wastes ~50% of bits.	High. Uses full range.	High. Activations use full range.
Weight Fit	Excellent. Weights are bell-shaped.	Good, but marginal gain.	Excellent.
Primary Use Case	Mobile/DSP (ARM SDOT)	Legacy (early TFLite)	Standard INT8 Inference

Conclusion: preparing for implementation

The journey through quantization choices reveals that there is no single "best" quantization. There is only the "best fit" for a specific constraint.

We will implement a custom Linear8bit layer in PyTorch. To keep our implementation realistic and performant, we will adopt the Hybrid Consensus: we will build a symmetric quantizer for our simulated weights and an affine (asymmetric) quantizer for our simulated inputs. This will allow us to observe the "Zero-Point Nudge" in action and verify the arithmetic precision against a floating-point baseline.

Let’s now proceed to build the tools that make these invisible grids visible.

2.5 End-to-end INT8 inference in PyTorch

We have built quantization from definitions and derivations. You now possess the equations for scale and zero-point, the geometric intuition for why affine mappings exist, and the vocabulary to distinguish granular error from overload error. What remains is to make these abstractions felt: to watch them operate on actual tensors, producing actual integers, and introducing actual distortion.

What follows provides that experience. We will implement the Hybrid Consensus (symmetric weights, asymmetric activations) in working PyTorch code, run inference through a quantized linear layer, and visually inspect how the integer grid captures, or fails to capture, the floating-point signal.

The goal is not to produce production-ready quantization code. It is to build intuition. By the end of this section, you should be able to look at a tensor histogram and predict whether symmetric or asymmetric quantization will waste bits. You should understand *why* the zero-point correction term appears in the arithmetic, not merely *that* it does.

We begin by translating sections 2.2 and 2.4 into code. The Hybrid Consensus prescribes two distinct quantization strategies: one for weights, one for activations. Each strategy reflects the statistical reality of its target tensor.

Weights emerge from gradient descent with L2 regularization. They cluster around zero in a roughly Gaussian distribution. For such data, symmetric quantization wastes nothing: the negative integers encode negative weights, the positive integers encode positive weights, and zero maps exactly to zero.

Listing 2.7 turns that constraint into code. The class exposes the same three operations as a textbook quantizer — calibrate, quantize and dequantize with two simplifications: the zero-point is hardwired to 0, and the integer range is locked to [−127, 127] to keep the grid perfectly symmetric. Watch how those two choices collapse the quantize equation from $q = round(r/S) + Z$ to a plain divide-and-round.

Listing 2.7 SymmetricQuantizer for weights

```
class SymmetricQuantizer:
    def __init__(self, bits=8):
        self.q_max = (1 << (bits - 1)) - 1
        self.q_min = -self.q_max
        self.scale = 1.0
        self.zero_point = 0

    def calibrate(self, tensor):
        abs_max = tensor.abs().max()
        self.scale = abs_max / self.q_max if abs_max > 0 else 1.0

    def quantize(self, tensor):
        scaled = tensor / self.scale
        rounded = torch.round(scaled)
        return torch.clamp(rounded, self.q_min, self.q_max).to(to

    def dequantize(self, q_tensor):
        return q_tensor.float() * self.scale
```

Three observations deserve attention. The range is [-127, 127], not [-128, 127]. This choice sacrifices one integer code to preserve perfect symmetry. The value -128 has no positive counterpart in signed 8-bit representation; excluding it ensures that *q_min = -q_max*, which simplifies the scale calculation to a single division by the absolute maximum. Calibration finds the absolute maximum. Because the grid is symmetric, we need only one boundary. The scale factor becomes *abs_max / 127*, ensuring that the largest magnitude in either direction lands exactly on the grid boundary. The quantize operation has no zero-point term. The equation reduces to $q = round(r / S)$. This is the computational simplification we sought in Section 2.2—no cross-terms, no runtime corrections, just division and rounding.

The affine quantizer (for activations)

Activations behave differently. After a ReLU, all values are non-negative. After a GeLU, most values are positive with a small negative tail. For these skewed distributions, symmetric quantization wastes the entire negative range of the integer grid.

Listing 2.8 implements that shift. Compared with the symmetric class in Listing 2.8, two things change: the integer range becomes [0, 255] (unsigned, since the data is non-negative), and the zero-point is computed from the data rather than fixed at zero. The zero-inclusion nudge inside `calibrate` is the line that keeps real 0.0 mapped to an exact integer — the constraint Section 2.2 argued was non-negotiable for ReLU sparsity and zero-padding.

Listing 2.8 AffineQuantizer for activations

```
class AffineQuantizer:
    def __init__(self, bits=8):
        self.q_min = 0
        self.q_max = (1 << bits) - 1
        self.scale = 1.0
        self.zero_point = 0

    def calibrate(self, tensor):
        r_min = tensor.min().item()
        r_max = tensor.max().item()
        r_min = min(r_min, 0.0)
        r_max = max(r_max, 0.0)

        real_range = r_max - r_min
        int_range = self.q_max - self.q_min

        if real_range == 0:
            self.scale, self.zero_point = 1.0, 0
        else:
            self.scale = real_range / int_range
            initial_z = self.q_min - (r_min / self.scale)
            self.zero_point = int(round(initial_z))
            self.zero_point = max(self.q_min, min(self.q_max, sel

    def quantize(self, tensor):
        scaled = (tensor / self.scale) + self.zero_point
        rounded = torch.round(scaled)
        return torch.clamp(rounded, self.q_min, self.q_max).to(to

    def dequantize(self, q_tensor):
        return self.scale * (q_tensor.float() - self.zero_point)
```

The critical difference appears in calibration. The range is [0, 255], unsigned. For strictly positive data, we use all 256 codes to represent positive values.

No bits are wasted on a negative range that the data never visits. The zero-point nudge is mandatory. It forces the calibrated range to include 0.0, even if the data's actual minimum is positive. This ensures that real zero maps to an exact integer, preserving sparsity from ReLU and enabling correct zero-padding in convolutions. The zero-point is calculated, not fixed. The formula $Z=q_{min}-(r_{min}/S)$ places the integer grid so that r_{min} maps to q_{min}. For ReLU outputs where $r_{min}=0$, this yields Z=0. For data shifted away from zero, Z takes a non-zero value that must be accounted for during inference.

The Linear8bit layer

With both quantizers defined, we can build a linear layer that performs the quantized arithmetic derived in Section 2.2. The key insight is that when weights use symmetric quantization ($Z_w=0$), the four-term expansion simplifies to two terms defined in Equation 2.9. The first term is the standard integer dot product. The second term corrects for the activation zero-point—and it can be precomputed because $\sum q_w$ depends only on the weights, which are fixed at inference time.

Listing 2.9 wires those pieces into a single `nn.Module`. The forward pass calibrates and quantizes the inputs and weights, computes the two surviving terms of Equation 2.9, rescales by $S_w \cdot S_x$, and adds the FP32 bias. Read it as a literal transcription of the math: every line corresponds to one symbol in Equation 2.9.

Listing 2.9 Linear8bit layer with hybrid quantization

```
class Linear8bit(nn.Module):
    def __init__(self, in_features, out_features, bias=True):
        super().__init__()
        self.weight = nn.Parameter(torch.randn(out_features, in_f
        self.bias = nn.Parameter(torch.randn(out_features)) if bi
        self.weight_quantizer = SymmetricQuantizer(bits=8)
        self.input_quantizer = AffineQuantizer(bits=8)

    def forward(self, x):
        self.input_quantizer.calibrate(x)
        x_q = self.input_quantizer.quantize(x)
```

```
        self.weight_quantizer.calibrate(self.weight)
        w_q = self.weight_quantizer.quantize(self.weight)

        term1 = torch.mm(x_q.float(), w_q.float().t())
        sum_w_q = torch.sum(w_q.float(), dim=1)
        term2 = self.input_quantizer.zero_point * sum_w_q

        y_int = term1 - term2
        scale = self.weight_quantizer.scale * self.input_quantize
        y_out = y_int * scale

        if self.bias is not None:
            y_out += self.bias
        return y_out
```

A few implementation notes. We cast integers to float for `torch.mm`. PyTorch's matrix multiplication does not natively support mixed int8/uint8 operands. In production runtimes like TensorRT or TFLite, dedicated integer kernels perform this operation directly in INT8 with INT32 accumulators. Our simulation captures the arithmetic correctly, even if it uses float operations underneath. The weight sum is precomputable. In a real deployment, this would happen once during model preparation and be stored alongside the quantized weights. The runtime cost is then a single scalar multiplication per output channel, not a reduction over the entire weight matrix. The bias stays in floating-point. Quantizing biases requires special handling (they accumulate in INT32 with scale $S_w \times S_x$). For clarity, we keep them in FP32 and add them after rescaling.

Running the experiment

We now have the machinery to test quantization on realistic data. The experiment in Listing 2.10 builds a `Linear8bit` layer and an FP32 reference `nn.Linear` with identical weights, then runs the same ReLU-shaped activations through both. The pair lets us compare the quantized output against an unimpeachable baseline rather than against itself.

Listing 2.10 Experiment setup

```
torch.manual_seed(42)
```

```
weights_fp32 = torch.randn(32, 64) * 0.1
inputs_fp32 = torch.relu(torch.randn(128, 64)) * 2.5

layer_8bit = Linear8bit(in_features=64, out_features=32)
layer_8bit.weight.data = weights_fp32.clone()

layer_fp32 = nn.Linear(64, 32, bias=True)
layer_fp32.weight.data = weights_fp32.clone()
layer_fp32.bias.data = layer_8bit.bias.data.clone()

output_fp32 = layer_fp32(inputs_fp32)
output_8bit = layer_8bit(inputs_fp32)
```

After execution, we can inspect the quantization parameters:

```
Weight Quantization (Symmetric):
  Scale: 0.003018
  Zero-Point: 0
  Range: [-127, 127]

Input Quantization (Affine):
  Scale: 0.041381
  Zero-Point: 0
  Range: [0, 255]
```

The weight scale is small (*0.003*) because the weights themselves are small (standard deviation 0.1). The input scale is larger (*0.041*) because ReLU outputs span a wider range after scaling by 2.5.

Crucially, both zero-points are zero in this particular run. Why? The weights are centered at zero, so symmetric quantization naturally has $Z_w=0$. The activations start at zero (ReLU's minimum), and the zero-point nudge sets $r_{min}=0$, which yields $Z_x=q_{min}-0/S=0$.

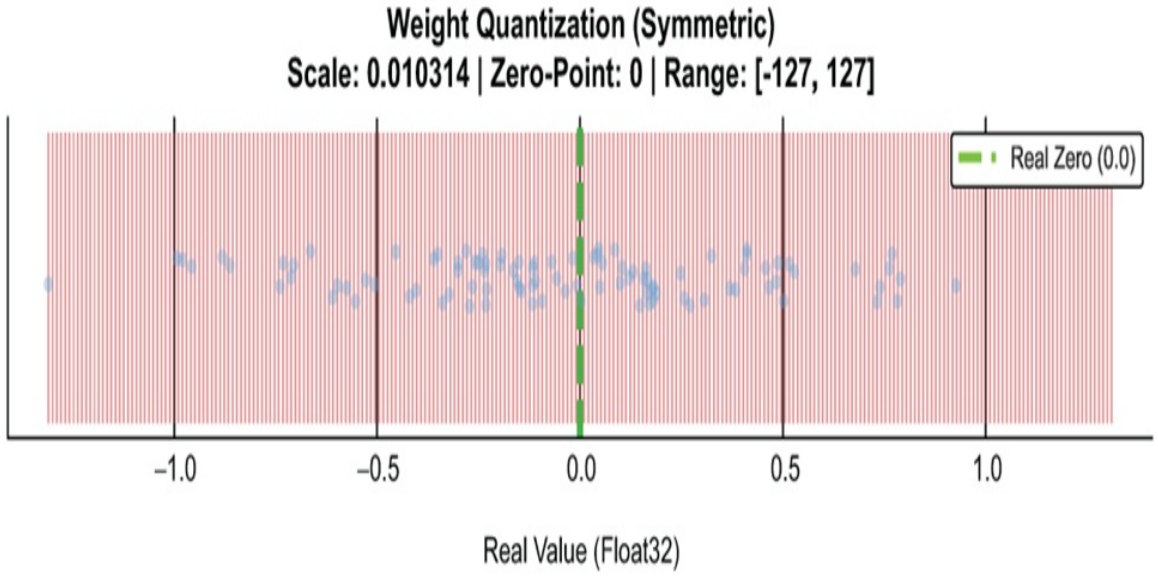
This is not always the case. If the activation distribution were shifted—say, ranging from 2.0 to 12.0—the affine quantizer would compute a non-zero Z_x to align the grid with the data.

Visualizing the integer grid

Numbers alone do not build intuition. Figure 2.7 plots the quantized weight

tensor against the integer grid it must inhabit. The grid is the same one Listing 2.8 derived from the data; the points are the weights from this run.

Figure 2.7 Weight quantization (symmetric)

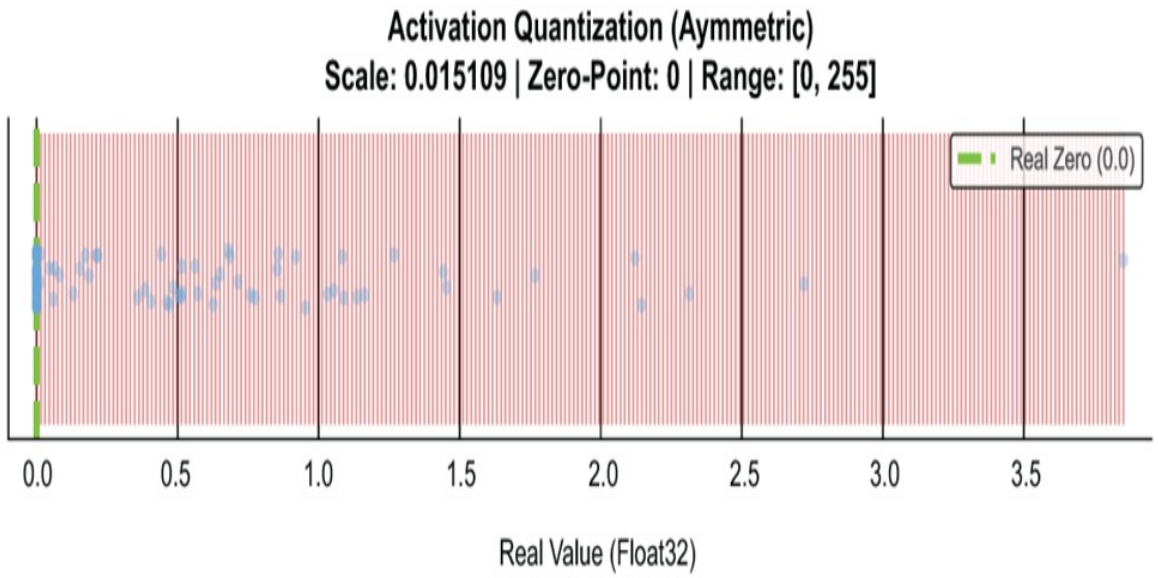


The figure shows the weight tensor (blue points) scattered across the real number line, with vertical red lines marking each integer grid point. The green dashed line indicates real zero, which aligns exactly with integer zero.

Several features deserve attention. The data forms a dense cloud around zero, with density falling off toward the tails—the expected Gaussian shape. The grid spacing (scale = 0.003) is fine enough to capture the variation. Points between adjacent grid lines will be rounded, but the distances are small. The grid extends symmetrically in both directions. Negative weights use negative integers; positive weights use positive integers. Nothing is wasted.

Figure 2.8 plots the same view for the activations. Because the activations are ReLU-floored, the picture is one-sided, and the integer grid has been re-anchored to match.

Figure 2.8 Activation quantization (affine/asymmetric)



All data points cluster on the positive side of the number line—this is ReLU output, after all. The grid has been configured to match: the range spans [0, 255] in integer space, mapped to [0.0, ~10.5] in real space. The green dashed line (real zero) aligns with integer zero. The zero-point nudge ensured this. No grid lines extend into negative territory. Every one of the 256 available codes represents a value the data might actually take.

This grid was calibrated from the data: the quantizer scanned the tensor, found r_max ≈ 10.55, and set S ≈ 0.04138. A different batch peaking at 8.0 would yield a tighter scale—finer resolution but clipping anything above 8.0. Unlike weights, activation ranges shift with input data, so calibration must be representative.

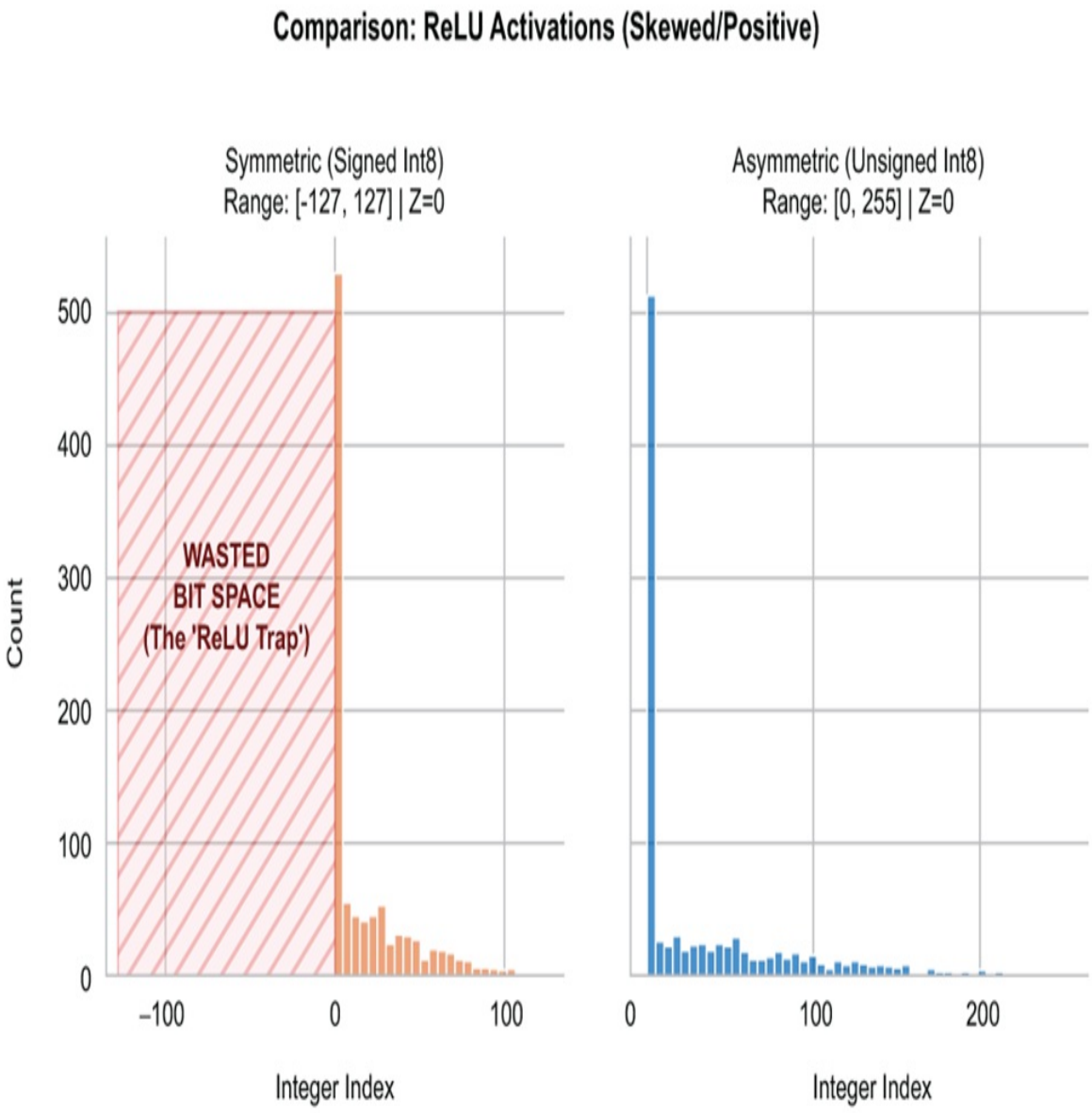
If we had used symmetric quantization here, half the grid would cover the range [-10.5, 0.0]—a region containing exactly zero data points. The step size would double, and granular error would increase proportionally.

The ReLU trap: symmetric vs. asymmetric

To make this trade-off visceral, we run both quantization strategies on identical ReLU data and compare how they allocate integer codes. Figure 2.9 puts the two outcomes side by side. The left panel shows where the integer codes land under the symmetric scheme; the right panel shows the same data

under the asymmetric scheme. The wasted half-grid on the symmetric side is the cost of ignoring the data's shape; the resolution doubling on the asymmetric side is what the Hybrid Consensus is reaching for.

Figure 2.9 Symmetric vs asymmetric on ReLU activations



The left panel shows symmetric quantization. The histogram of assigned integers clusters entirely in the positive range [0, 127]. The shaded red region on the negative side—integers [-127, -1]—is labeled "WASTED BIT SPACE." These 127 codes are allocated but never used. Effectively, we have

7-bit quantization with an 8-bit storage cost.

The right panel shows asymmetric quantization. The histogram spreads across the full range [0, 255]. Every code can potentially be assigned. The resolution doubles compared to the symmetric case.

The numerical impact:

```
Symmetric Stats:  Scale = 0.139, Range Used = [0, 127]
Asymmetric Stats: Scale = 0.069, Range Used = [0, 255]
Resolution Gain:  Asymmetric is 2.01x more precise
```

A scale of 0.069 means each integer step represents 0.069 units in real space. A scale of 0.139 means each step represents 0.139 units—twice as coarse. For the same 8-bit storage, asymmetric quantization delivers twice the precision on ReLU activations.

This is the ReLU Trap. It catches anyone who applies symmetric quantization uniformly without inspecting the data distribution. The fix is simple: use the Hybrid Consensus. But recognizing when you have fallen into the trap requires looking at histograms.

Verifying the arithmetic

The ultimate test is numerical: does our quantized layer produce the same outputs as the floating-point baseline?

Listing 2.11 settles it by subtracting the INT8 output from the FP32 reference element-wise and reporting two scalar summaries: mean squared error (sensitive to large per-element errors) and mean absolute error (a more interpretable average deviation).

Listing 2.11 Error analysis

```
error_tensor = output_8bit - output_fp32
mse = torch.mean(error_tensor**2).item()
mae = torch.mean(torch.abs(error_tensor)).item()

print(f"Mean Squared Error (MSE): {mse:.8f}")
print(f"Mean Absolute Error (MAE): {mae:.8f}")
```

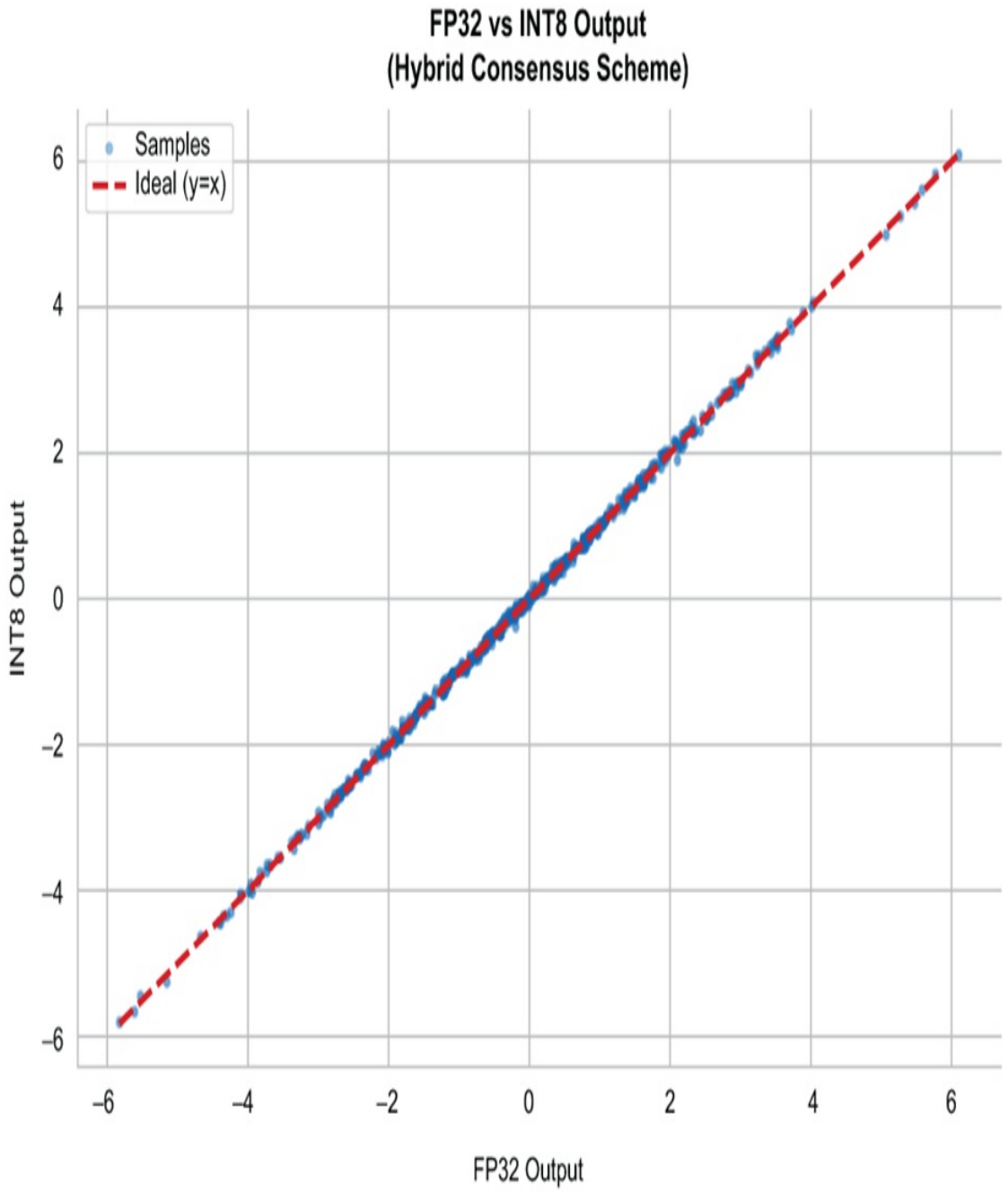
Output:

```
Mean Squared Error (MSE): 0.00019618
Mean Absolute Error (MAE): 0.01111147
```

The errors are small—small enough that for many applications, INT8 inference is indistinguishable from FP32. But they are not zero. Quantization always introduces error; the question is whether that error matters for the task at hand.

Numbers tell us the *size* of the error. Figure 2.10 tells us its *shape*. Each point in the scatter plots one output element's FP32 value against its INT8 counterpart, with the identity line drawn for reference. The picture answers a different question from MSE alone: is the error random and centered, or biased in some direction?

Figure 2.10 FP32 baseline vs INT8 output



The scatter plot tells the story at a glance. Each point represents one output element: its x-coordinate is the FP32 baseline value, its y-coordinate is the INT8 quantized value. The red dashed line is the identity—perfect agreement.

What we see is remarkably tight clustering along the diagonal. Points hug the identity line across the entire output range, from -6 to +6. There is no visible bias (the cloud does not drift above or below the line), and no visible

widening of the scatter at the extremes—the error is uniform across the output range. The Hybrid Consensus has done its job: the quantized arithmetic reproduces the floating-point computation to within the noise floor of 8-bit representation.

If this plot showed points scattered widely around the diagonal, or systematically offset from it, we would know something had gone wrong—perhaps the calibration data was unrepresentative, or outliers stretched the scale beyond what the majority of values needed.

Watching the zero-point correction earn its keep

Our ReLU experiment hides the cross-term. With *r_min* = 0, the affine quantizer set *Z_x* = 0, and term 2 of Equation 2.9 contributed nothing — ignoring it would still produce the right answer. That makes the demo a poor argument for *why we wrote term 2 at all*. To see the correction matter, we re-run the layer on activations that don't start at zero, as you would find after a LayerNorm, an attention softmax, or any operator whose output is shifted away from the origin.

Listing 2.12 runs the layer twice on the same shifted inputs: once through the correct `forward` (which subtracts term 2) and once through a naive path that drops term 2 entirely. Comparing both to the FP32 reference tells us how much of the layer's accuracy depends on the correction we wrote down in Section 2.2.

Listing 2.12 Shifted activations: the correction stops being optional

```
torch.manual_seed(123)
inputs_shifted = torch.randn(128, 64) * 2.0 + 5.0       #A

layer_shifted = Linear8bit(in_features=64, out_features=32)
layer_shifted.weight.data = weights_fp32.clone()

layer_shifted_fp32 = nn.Linear(64, 32, bias=True)
layer_shifted_fp32.weight.data = weights_fp32.clone()
layer_shifted_fp32.bias.data = layer_shifted.bias.data.clone()

output_fp32_ref = layer_shifted_fp32(inputs_shifted)
output_correct  = layer_shifted(inputs_shifted)         #B
```

```
# Naive: drop the term2 = Zx · ∑q_w correction
x_q = layer_shifted.input_quantizer.quantize(inputs_shifted)
w_q = layer_shifted.weight_quantizer.quantize(layer_shifted.weigh
term1 = torch.mm(x_q.float(), w_q.float().t())
scale = (layer_shifted.weight_quantizer.scale *
         layer_shifted.input_quantizer.scale)
output_naive = term1 * scale + layer_shifted.bias       #C

mae_correct = (output_correct - output_fp32_ref).abs().mean()
mae_naive   = (output_naive   - output_fp32_ref).abs().mean()

print(f"Zero-point Zx:       {layer_shifted.input_quantizer.zero_
print(f"MAE vs FP32 (correct): {mae_correct.item():.4f}")
print(f"MAE vs FP32 (naive):   {mae_naive.item():.4f}")
```

The output:

```
Zero-point Zx:         48
MAE vs FP32 (correct): 0.0302
MAE vs FP32 (naive):   1.4174
Ratio (naive/correct): 46.9x
```

There are two things to notice.

First, dropping the correction blows the per-element error up by nearly 50×, and the inflated error is not noise around the right answer — it is a systematic bias. Every output element loses the same $Z\_x \cdot \sum q\_w$ term, and that bias does not average out across the batch; it accumulates.

Second, the cost of *adding* term 2 back is negligible: ∑q_w depends only on the weights, which are fixed at inference time, so the whole correction can be precomputed once and stored alongside the quantized weights. The runtime cost is one scalar-vector multiply per layer.

That is the bargain asymmetric quantization makes. Symmetric quantization suits centered data — weights, with their Gaussian distributions around zero, lose nothing from a symmetric grid, and the computational reward is significant: no cross-terms, no runtime corrections, just integer dot products. Asymmetric quantization suits skewed or shifted data — activations from ReLU, GeLU, or post-norm operators benefit from a shifted grid, and the zero-point nudge keeps real zero exactly representable. The Hybrid

Consensus combines both, and the cross-term correction is the price you pay for the asymmetric side of that combination.

Visualization is diagnostic. A histogram of your data, overlaid with the quantization grid, reveals wasted bits immediately. If half the grid covers empty space, you have chosen the wrong scheme. And the arithmetic is exact within rounding: the equations from Section 2.2 produce outputs that match FP32 to within quantization noise — as long as you actually compute every term.

Reproducing these results: the scripts used here are available here. Representative blocks appear in the chapter; the full runnable versions live in the repo.

2.6 Summary

- Fixed-point arithmetic stores numbers on a rigid discrete grid determined at design time. The Q-format notation makes explicit the fundamental bargain: allocate bits to range or to precision, but never both. Two's complement representation enables negative numbers through overflow, while saturation at the overflow cliff constrains every fixed-point system.
- The affine transformation $r = S(q - Z)$ bridges floating-point tensors and integer storage. Scale (S) is the ratio of real range to integer range; zero-point (Z) aligns real zero with an integer coordinate. The zero-point nudge forces the calibrated range to include 0.0, preserving sparsity after ReLU and enabling correct zero-padding.
- Granular error arises when values land between grid points and must be rounded, bounded by $S/2$. Overload error arises when values fall outside the grid and are clipped, introducing unbounded distortion. Widening the grid reduces clipping but increases rounding; tightening it does the reverse.
- Symmetric quantization centers the grid on zero, wasting capacity on one-sided distributions but eliminating cross-term overhead. Asymmetric quantization shifts the grid to follow data, reclaiming bits at the cost of runtime correction. The Hybrid Consensus—symmetric weights, asymmetric activations—matches each scheme to its statistical

reality.

- Visualization reveals wasted bits immediately: if half the integer codes cover empty space, the wrong scheme was chosen. The quantized arithmetic, when implemented correctly with zero-point correction, matches FP32 output to within the noise floor of 8-bit representation.
- Calibration is not a one-time event. Scale and zero-point derivations in this chapter assumed known min/max—true for weights but not activations, whose ranges shift with every batch. Strategies like percentile clipping and entropy minimization replace raw min/max with robust boundaries; when input distributions drift, recalibration may be needed.

3 Choosing What to Quantize and at What Granularity

This chapter covers

- Weight, activation, and KV cache quantization targets
- Per-tensor, per-channel, and group-wise granularity schemes
- Memory layout and kernel efficiency trade-offs
- Decision frameworks for precision allocation

A neural network is not a single tensor: it contains weights (the parameters learned during training, frozen at inference), activations (the intermediate tensors that flow between layers as data passes through, recomputed for every input), and, in transformers, a key-value cache (the KV cache: the keys K and values V from past tokens, stored so the attention mechanism doesn't have to recompute them at every generation step).

These have radically different statistical properties and sensitivities to error.

Weights are static and bell-shaped, tolerant of rounding; activations shift with every input and develop extreme outliers in transformer hidden states; the KV cache sits somewhere in between, written once per token but read thousands of times. Quantizing all three identically—say, INT8 symmetric with a single scale per tensor—wastes precision on weights that could tolerate INT4, fails to capture activation outliers that need special handling, and ignores the structural asymmetry between keys and values in the cache. Quantizing none is equally wasteful: a 70B-parameter model in FP16 occupies 140 GB, most of it dead weight during memory-bound autoregressive generation.

The skill lies in knowing where each bit of precision pays for itself.

Beyond choosing *what* to quantize, you must choose *at what granularity*. A single scale per tensor is simple but crude; a scale per output channel is more accurate; group-wise schemes split the difference. The right choice depends

on the tensor's statistics, the target hardware, and acceptable accuracy loss.

Our goal here is to help you build that judgment, systematically. We start by examining what makes weights, activations, and the KV cache fundamentally different as quantization targets, then work through the granularity decisions —per-tensor, per-channel, per-token, and group-wise—that determine how much precision each component actually needs.

3.1 Identify where precision loss matters

A neural network contains weights (learned parameters, frozen after training), activations (intermediate values computed during inference), and—in transformers—a KV cache (attention memory that grows with sequence length). Understanding their distinct sensitivities to precision loss lets you reason about trade-offs instead of memorizing rules.

Because weights don't change at inference time, we can analyze their distribution once, offline, with no need to estimate ranges from running data or worry about distribution shifts across inputs. Let's look at what a typical weight tensor actually contains.

Listing 3.1 Inspecting weight distribution

```
model = torch.hub.load('pytorch/vision', 'resnet18', pretrained=T
conv1_weights = model.conv1.weight.data
print(f"Shape: {conv1_weights.shape}")
print(f"Min: {conv1_weights.min().item():.4f}")
print(f"Max: {conv1_weights.max().item():.4f}")
print(f"Mean: {conv1_weights.mean().item():.6f}")
print(f"Std: {conv1_weights.std().item():.4f}")
```

Output:

```
Shape: torch.Size([64, 3, 7, 7])
Min: -0.8434 | Max: 1.0165 | Mean: 0.000029 | Std: 0.1297
```

Notice the near-zero mean.

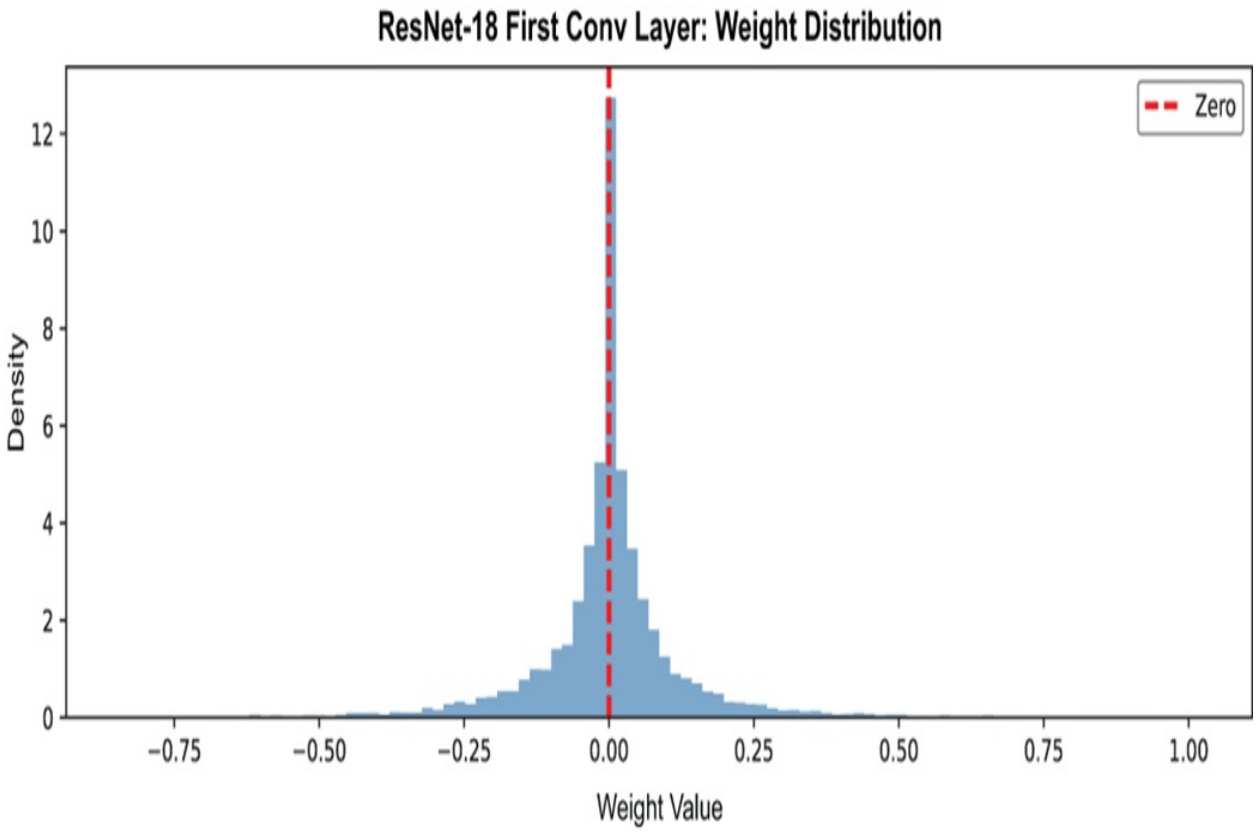
Modern neural networks are trained with regularization—typically L2 weight

decay—that penalizes large weights. This regularization, combined with the dynamics of gradient descent, produces weight distributions that cluster around zero in a roughly bell-shaped form. Empirically, conv weights tend toward Gaussian (light tails); transformer linear weights skew Laplacian (heavier tails). The distinction matters for range estimation: a Laplacian distribution puts more mass in the tails, so percentile clipping is more aggressive than max-based scaling. This symmetry is common but not universal—models trained without weight decay, with certain normalization schemes, or with sparse activation functions can develop skewed or bimodal weight distributions.

When symmetry doesn't hold, asymmetric quantization (using a non-zero zero-point) recovers the lost precision. We can visualize this directly by plotting the histogram of conv1's weights.

The companion script ch3/3.1 resnet18_dist.ipynb generates the distribution plot shown in Figure 3.1. This confirms the characteristic bell-shaped distribution we'd expect from a regularized network.

Figure 3.1 ResNet-18 first conv layer weight distribution



The resulting histogram shows a symmetric distribution centered at zero—precisely the shape that symmetric quantization handles optimally. The negative weights need negative integers; the positive weights need positive integers; zero maps to zero. No bits are wasted.

Quantization adds a bounded perturbation to each weight. For an INT8 quantizer with scale S, the granular error is bounded by $|\text{error}| \leq S/2$ (overload error can be avoided entirely for weights since range boundaries are known).

Because weights emerge from stochastic training with noise, dropout, and augmentation, they tolerate small variations. Wu et al. (2020) measured weight-only INT8 quantization on a representative set of vision models and found accuracy loss under 0.5% on ImageNet across ResNet, MobileNet, and EfficientNet families (we'll see the table-by-table numbers in Table 3.2).

The memory win: linear reduction

Quantizing weights delivers a direct, linear reduction in model size. A 7-billion parameter model stored in FP16 (2 bytes per parameter) occupies 13 GB. The same model in INT8 (1 byte per parameter) occupies 6.5 GB. And in INT4 (0.5 bytes per parameter) just 3.25 GB.

This isn't just a storage saving—it's a bandwidth saving. As Chapter 1 established, modern inference is memory-bound. Every byte saved is a byte that doesn't need to travel from DRAM to the compute unit. The following calculation shows the concrete memory impact for a 7B-parameter model.

Listing 3.2 Model size by precision

```
def calculate_model_size(num_params, bytes_per_param):
    return (num_params * bytes_per_param) / (1024**3)           #

num_params = 7e9
for dtype, bpp in [('FP32', 4), ('FP16', 2), ('INT8', 1), ('INT4'
    size = calculate_model_size(num_params, bpp)
    print(f"{dtype}: {size:.1f} GB")
```

Output:

```
FP32: 26.1 GB | FP16: 13.0 GB | INT8: 6.5 GB  | INT4: 3.3 GB
```

Despite these favorable properties, some weights resist quantization. Embedding tables have different statistics than linear layers—individual token rows can have high variance. Classification heads directly produce logits fed to softmax, so small perturbations flip predictions; practitioners often keep them in higher precision. Bias vectors are tiny tensors where outliers dominate min/max estimates; they're typically left in FP32 since their memory footprint is negligible.

Activations: the dynamic challenge

If weights are the frozen foundation, activations are the living signal—the intermediate tensors produced as data flows through each layer, recomputed fresh for every input. This dynamism makes activation quantization fundamentally harder.

The distribution of activations changes with every input. An image of a cat produces different intermediate representations than an image of a car. A prompt asking about physics produces different hidden states than a prompt asking about cooking.

We can measure this variation directly using forward hooks.

Listing 3.3 Capturing activation statistics across inputs

```
model = nn.Sequential(
    nn.Linear(10, 32),
    nn.ReLU(),
    nn.Linear(32, 10)
)

activations = {}
def hook_fn(name):
    def hook(module, input, output):
        activations[name] = output.detach()
    return hook

model[1].register_forward_hook(hook_fn('relu_output'))          #
input1 = torch.randn(1, 10)
input2 = torch.randn(1, 10) * 3                                 #
_ = model(input1)
act1 = activations['relu_output'].clone()
_ = model(input2)
act2 = activations['relu_output'].clone()
print(f"Input 1 - Mean: {act1.mean():.4f}, Max: {act1.max():.4f}"
print(f"Input 2 - Mean: {act2.mean():.4f}, Max: {act2.max():.4f}"
```

Output:

```
Input 1 - Mean: 0.2588, Max: 1.3093 | Input 2 - Mean: 0.8258, Max
```

The same layer produces activations with 3× different maximum values depending on the input. If we calibrated our quantizer on data similar to input1, we would clip values from input2. If we calibrated on input2, we would waste resolution on input1.

The outlier problem

Activations often exhibit heavy-tailed distributions. Most values cluster in a reasonable range, but a small fraction spike to extreme values. This is especially pronounced in transformers, where certain "outlier channels" in the hidden states can be 10–100× larger than typical channels. Listing 3.4 shows how dramatically a few outliers distort the quantization scale.

Listing 3.4 Outlier impact on scale

```
normal_activations = np.random.randn(1000) * 0.5                #
outlier_activations = np.random.randn(10) * 50                  #
all_activations = np.concatenate([normal_activations, outlier_act
scale_with_outliers = np.abs(all_activations).max() / 127
scale_without_outliers = np.percentile(np.abs(all_activations), 9
print(f"Scale with outliers: {scale_with_outliers:.4f}")
print(f"Scale at 99th pct: {scale_without_outliers:.4f}")
print(f"Resolution ratio: {scale_with_outliers/scale_without_outl
```

Output:

```
With outliers: 0.7222 | 99th percentile: 0.0115
Resolution ratio: 63.0x coarser
```

A handful of outliers make the scale factor 60× larger than necessary for 99% of the data—as Listing 3.4 just showed. The core challenge: protect the outliers (destroying resolution for everything else) or clip them (accepting unbounded error for a few values)?

Calibration: the necessary overhead

Because activation statistics are input-dependent, we can't quantize activations purely offline. We need calibration: running representative data through the network to estimate the ranges that activations will occupy in production. Listing 3.5 implements a minimal observer that tracks the running min and max across calibration batches and derives the scale and zero-point from those bounds, with a symmetric variant for INT8 hardware and an asymmetric variant for unsigned UINT8.

Listing 3.5 Calibration observer for activation ranges

```
class ActivationObserver:
    def __init__(self):
        self.min_val = float('inf')
        self.max_val = float('-inf')

    def observe(self, tensor):
        self.min_val = min(self.min_val, tensor.min().item())   #
        self.max_val = max(self.max_val, tensor.max().item())

    def get_scale_zeropoint(self, bits=8, symmetric=False):
        if symmetric:
            abs_max = max(abs(self.min_val), abs(self.max_val))
            q_max = (1 << (bits - 1)) - 1
            scale = abs_max / q_max
            zero_point = 0
        else:
            q_min, q_max = 0, (1 << bits) - 1
            scale = (self.max_val - self.min_val) / (q_max - q_mi
            zero_point = int(round(q_min - self.min_val / scale))
        return scale, zero_point
```

Calibration introduces a new failure mode: distribution mismatch. If the calibration data doesn't represent production traffic, the estimated ranges will be wrong.

The compute vs. memory trade-off

Unlike weight quantization, which primarily saves memory, activation quantization affects both memory and compute. Activations are the operands of matrix multiplications. If both weights and activations are quantized, the entire GEMM can execute in integer arithmetic—INT8 weights × INT8 activations with INT32 accumulate—using half the memory bandwidth and achieving higher throughput on hardware with integer tensor cores.

But this only works if both operands are quantized. Quantizing weights alone still requires dequantizing to FP16 before the multiply—saving memory but not compute. This is why activation quantization, despite its difficulty, is essential for maximum speedup.

The KV cache: transformers' memory bottleneck

The key-value cache is a transformer-specific data structure that has become the dominant memory bottleneck in LLM inference.

During the attention mechanism, each token's representation is projected into keys (K) and values (V) that allow other tokens to attend to it. In autoregressive generation, we produce one token at a time. Without caching, we would need to recompute the K and V projections for all previous tokens at every generation step—quadratic in sequence length.

The KV cache solves this by storing the K and V tensors for all previously generated tokens. At each new step, we only compute K and V for the new token and append them to the cache.

A simplified implementation shows the core mechanism.

Listing 3.6 Simplified KV cache implementation

```
class CachedAttention:
    def __init__(self, d_model, n_heads):
        self.d_head = d_model // n_heads
        self.W_k = torch.randn(d_model, d_model) * 0.02
        self.W_v = torch.randn(d_model, d_model) * 0.02
        self.k_cache = None
        self.v_cache = None

    def forward(self, x):
        k = x @ self.W_k
        v = x @ self.W_v
        if self.k_cache is None:
            self.k_cache, self.v_cache = k, v
        else:
            self.k_cache = torch.cat([self.k_cache, k], dim=1)
            self.v_cache = torch.cat([self.v_cache, v], dim=1)
        scores = torch.matmul(x @ self.W_q, self.k_cache.T)
        attn = torch.softmax(scores / math.sqrt(self.d_head), dim
        return torch.matmul(attn, self.v_cache)
```

Why the KV cache dominates memory

For large models with long contexts, the KV cache can exceed the model weights in memory consumption. For a transformer with L layers, H attention

heads, D head dimension, S sequence length, and B batch size, the KV cache stores $2 \times L \times H \times D \times S \times B$ elements (the factor of 2 accounts for both K and V). Plugging in Llama-2 70B's architecture parameters reveals how quickly this grows.

Listing 3.7 KV cache size calculation

```
def kv_cache_size_gb(num_layers, num_heads, head_dim, seq_length,
                     batch_size, bytes_per_element=2):
    total_bytes = 2 * num_layers * num_heads * head_dim * \
                  seq_length * batch_size * bytes_per_element
    return total_bytes / (1024**3)

llama_70b = dict(num_layers=80, num_heads=64, head_dim=128)
model_size_fp16 = 140  # GB
for seq_len in [2048, 8192, 32768, 131072]:
    kv_size = kv_cache_size_gb(**llama_70b, seq_length=seq_len, b
    ratio = kv_size / model_size_fp16 * 100
    print(f"Seq {seq_len:>6}: {kv_size:>6.2f} GB ({ratio:>5.1f}%
```

Output:

```
  Seq   2048:   5.00 GB (  3.6% of weights) | Seq   8192:  20.00
  Seq  32768:  80.00 GB ( 57.1% of weights) | Seq 131072: 320.00
```

At 128K context length—now common in frontier models—the KV cache exceeds the model weights in size. For a batch size of 8 (modest for a production server), multiply these numbers by 8. This is why KV cache quantization is not optional for long-context LLMs.

Why KV cache quantization is different

The KV cache has properties that make it neither like weights nor like typical activations. Each entry is written once when its token is processed, then read on every subsequent generation step—a position-0 entry in a 4096-token generation gets read 4,095 times. This amplifies dequantization overhead but makes write-time costs negligible. KV cache access is almost purely memory-bound, so quantization here is about memory savings, not compute speedup.

Research has shown that certain hidden dimensions consistently produce larger K/V values regardless of input. These "outlier channels" suggest per-channel quantization is especially effective for KV caches. Because the cache feeds into softmax-normalized attention, errors compound subtly—a small key error shifts the attention distribution over all values.

Table 3.1 lays the three targets side by side across the properties that drive their quantization strategies: when statistics become available, how stable they are, where outliers live, and what each one actually buys you.

Table 3.1 Quantization target comparison

Property	Weights	Activations	KV Cache
When computed	Training (frozen)	Every forward pass	Once per token
Distribution stability	Static	Input-dependent	Token-dependent, channel patterns stable
Typical shape	Zero-centered, bell-shaped	Varies by activation fn	Varies by layer depth
Outlier behavior	Rare	Common in transformers	Consistent outlier channels
Quantization difficulty	Easy	Hard	Medium
Memory impact	Linear in parameters	Transient	Linear in seq_len × batch
Primary benefit	Model size reduction	Compute speedup	Memory for long contexts
Risk of accuracy loss	Low	Medium	Medium (attention-dependent)

Not all tensors within each category are equally sensitive. Some layers in a network contribute more to the final output than others. Quantizing them aggressively causes disproportionate accuracy loss.

The standard way to identify sensitive layers is perturbation analysis: quantize one layer at a time, measure accuracy, and rank layers by their

impact. The following function automates this process for any model.

Listing 3.8 Layer sensitivity via perturbation analysis

```
def measure_layer_sensitivity(model, test_fn, layer_names):
    baseline_acc = test_fn(model)
    sensitivities = {}
    for name in layer_names:
        quantized_model = deepcopy(model)
        for n, module in quantized_model.named_modules():
            if n == name and hasattr(module, 'weight'):
                with torch.no_grad():
                    w = module.weight
                    scale = w.abs().max() / 127
                    w_quant = torch.round(w / scale) * scale
                    module.weight.copy_(w_quant)
        sensitivities[name] = baseline_acc - test_fn(quantized_mo
    return sensitivities
```

Common sensitivity patterns

Across architectures, certain patterns emerge. First and last layers are more sensitive than middle layers—the first interfaces with raw input, the last produces final predictions. Attention layers tend to be more sensitive than feedforward layers because softmax creates a winner-take-all dynamic. Layers with small weight ranges are more sensitive because the smaller scale factor makes quantization noise relatively larger.

NOTE

A useful mental model for the rest of the chapter: each tensor type has a different read-to-write ratio across an inference. Weights are written once at training time and read every forward pass. Activations are written and read once per forward pass. KV cache entries are written once per token but read once per subsequent generation step, so a position-0 entry in a 4096-token completion is read 4,095 times. The ratio drives where quantization pays for itself, which is why the KV cache became the highest-leverage target for sub-8-bit precision in long-context inference.

Knowing what to quantize is only half the problem. The next question is how

finely to tune the quantization parameters within each target. We begin with weights—the easiest target—and the choice that has the largest practical impact on weight quantization quality: per-tensor versus per-channel granularity.

3.2 Quantize weights with per-tensor and per-channel choices

The two dominant choices are per-tensor quantization (one scale factor for the entire weight matrix) and per-channel quantization (one scale per output channel). We'll build both, visualize the difference, and develop the judgment to choose between them.

Per-tensor quantization: the simplest baseline

We compute a single scale factor from the entire weight tensor and apply it uniformly to every element.

For a weight tensor W with shape [out_features, in_features], per-tensor symmetric quantization computes $S=max(|W|)/q_{max}$ where $q_{max}=127$ for INT8. Every weight is then quantized as $q=round(w/S)$ and dequantized as $\hat{w}=q \cdot S$.

Listing 3.9 translates this into a reusable class:

Listing 3.9 Per-tensor symmetric quantizer

```
class PerTensorQuantizer:
    def __init__(self, bits=8):
        self.q_max = (1 << (bits - 1)) - 1
        self.q_min = -self.q_max
        self.scale = None

    def calibrate(self, tensor):
        abs_max = tensor.abs().max()
        self.scale = abs_max.item() / self.q_max if abs_max > 0 e

    def quantize(self, tensor):
        scaled = tensor / self.scale
```

```
        rounded = torch.round(scaled)
        return torch.clamp(rounded, self.q_min, self.q_max).to(to

    def dequantize(self, q_tensor):
        return q_tensor.float() * self.scale

    def quantize_dequantize(self, tensor):
        self.calibrate(tensor)
        return self.dequantize(self.quantize(tensor))
```

Let's apply this to a real model. We'll use ResNet-18 as our running example—it's small enough to iterate quickly but representative of production CNN architectures.

Listing 3.10 Per-tensor quantization of ResNet-18 conv1

```
model = torch.hub.load('pytorch/vision', 'resnet18', weights='IMA
layer = model.conv1
weight = layer.weight.data.clone()
print(f"Weight shape: {weight.shape}")
print(f"Total elements: {weight.numel():,}")

quantizer = PerTensorQuantizer(bits=8)
weight_reconstructed = quantizer.quantize_dequantize(weight)

error = weight - weight_reconstructed
mse = (error ** 2).mean().item()
max_error = error.abs().max().item()
print(f"Scale: {quantizer.scale:.6f}")
print(f"MSE: {mse:.8f}")
print(f"Max absolute error: {max_error:.6f}")
```

Output:

```
Layer: conv1
Weight shape: torch.Size([64, 3, 7, 7]) | [out_channels, in_chann
Total elements: 9,408
Scale: 0.008 | MSE: 0.00000466 | Max absolute error: 0.004 | Erro
```

The numbers look good—sub-1% error relative to the weight range. But these aggregate statistics hide a critical problem.

The outlier problem

Per-tensor quantization fails when different parts of the tensor have different scales. Let's examine why by looking at the per-channel statistics:

Listing 3.11 Per-channel utilization analysis

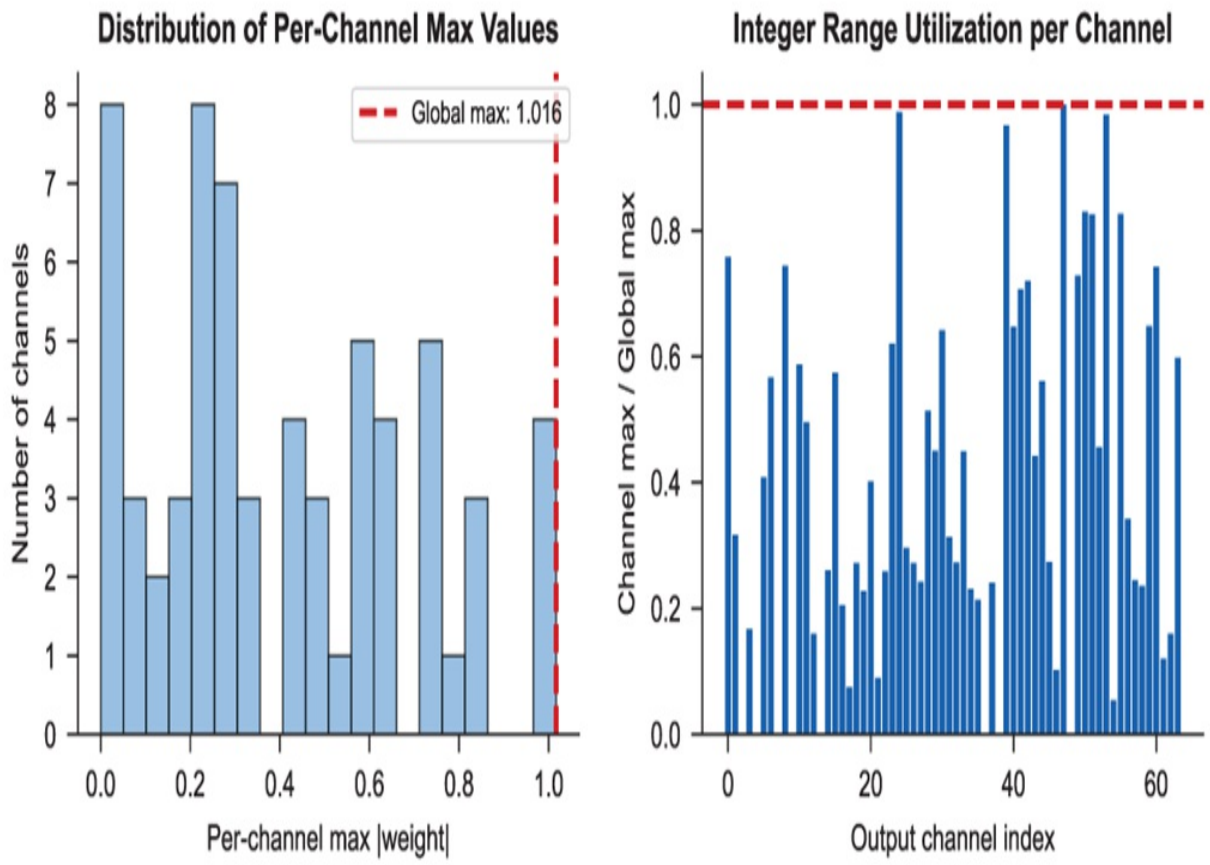
```
weight_2d = weight.view(weight.shape[0], -1)
channel_abs_max = weight_2d.abs().max(dim=1).values
global_abs_max = weight.abs().max().item()
utilization = channel_abs_max / global_abs_max
print(f"Min utilization: {utilization.min():.2%}")
print(f"Mean utilization: {utilization.mean():.2%}")
print(f"Channels using <50% of range: {(utilization < 0.5).sum().
```

Output:

```
Min: 0.00% | Max: 100.00% | Mean: 39.89% | Channels using <50% of
```

To see why per-tensor quantization wastes precision, let's examine the weight distribution in ResNet-18's first convolution. Figure 3.2 plots the per-channel maximum weights alongside their integer range utilization.

Figure 3.2 Most channels in ResNet-18's conv1 have maximum weights well below the global max of 1.016 (left), using only 20–50% of the integer range under per-tensor quantization (right)—the inefficiency that per-channel quantization eliminates.



This reveals a striking limitation. A single global scale factor compresses smaller channels into a fraction of the integer range. The mean utilization is only 39.89%—effectively losing more than 1 bit of precision across the entire tensor due to the outlier penalty.

Per-channel quantization: one scale per output

The solution is obvious once you see the problem: give each channel its own scale factor, tuned to its own distribution.

For a weight tensor W with shape [out_features, in_features], per-channel quantization computes a separate scale for each output channel: $S_c = max(|W_c|) / q_{max}$ where W_c is the slice of weights corresponding to output channel c. Each channel is then quantized with its own scale: $q_c = round(w_c / S_c)$ and dequantized: $\hat{w}_c = q_c \cdot S_c$. The implementation handles the broadcasting needed to apply per-channel scales to multi-dimensional

tensors.

Listing 3.12 Per-channel symmetric quantizer

```
class PerChannelQuantizer:
    def __init__(self, bits=8, channel_axis=0):
        self.q_max = (1 << (bits - 1)) - 1
        self.q_min = -self.q_max
        self.channel_axis = channel_axis
        self.scales = None

    def calibrate(self, tensor):
        num_channels = tensor.shape[self.channel_axis]
        tensor_2d = tensor.reshape(num_channels, -1)
        channel_abs_max = tensor_2d.abs().max(dim=1).values
        channel_abs_max = torch.where(
            channel_abs_max == 0,
            torch.ones_like(channel_abs_max),
            channel_abs_max
        )
        self.scales = channel_abs_max / self.q_max

    def quantize(self, tensor):
        num_channels = tensor.shape[self.channel_axis]
        scale_shape = [1] * tensor.ndim
        scale_shape[self.channel_axis] = num_channels
        scales_broadcast = self.scales.view(scale_shape)

        scaled = tensor / scales_broadcast
        return torch.clamp(torch.round(scaled), self.q_min, self.

    def dequantize(self, q_tensor):
        num_channels = q_tensor.shape[self.channel_axis]
        scale_shape = [1] * q_tensor.ndim
        scale_shape[self.channel_axis] = num_channels
        return q_tensor.float() * self.scales.view(scale_shape)
```

Comparing per-tensor vs. per-channel

Now let's run both quantizers on the same weight tensor and compare:

Listing 3.13 Per-tensor vs per-channel comparison

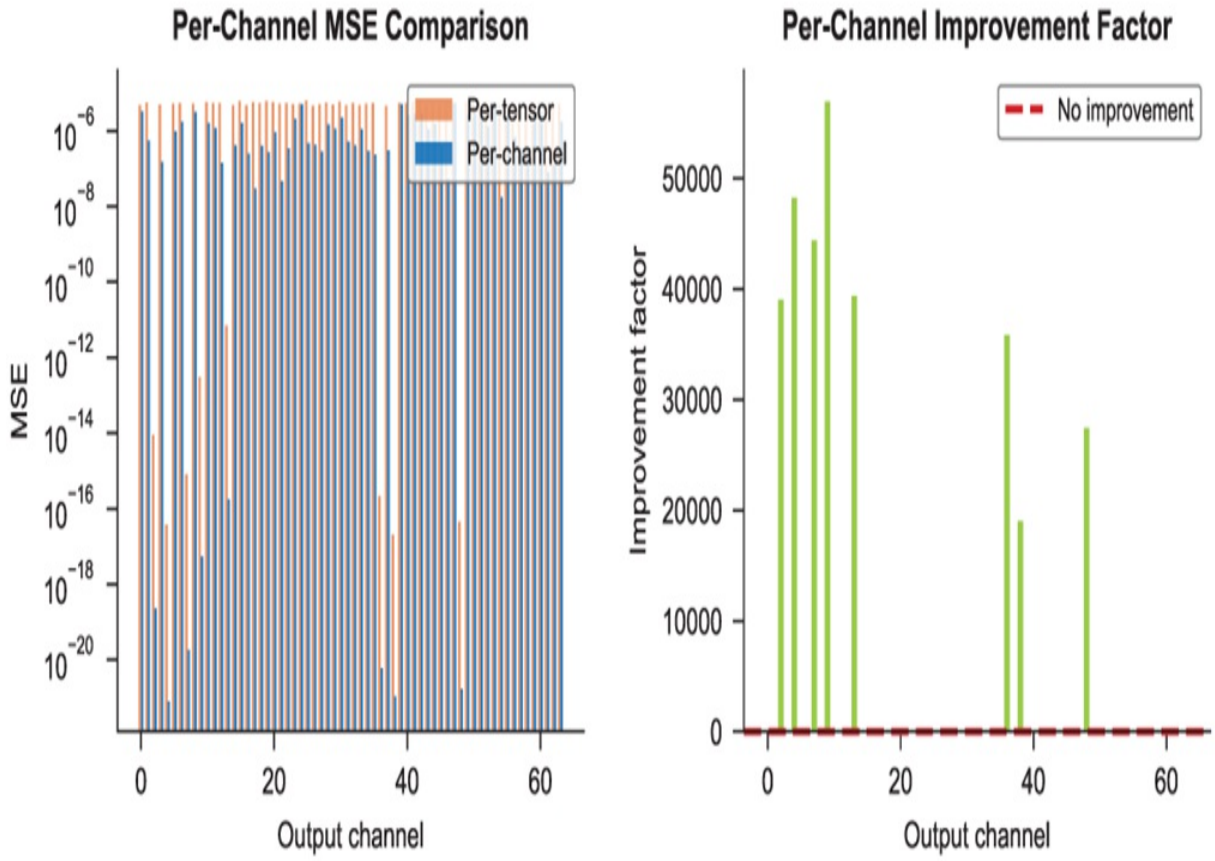
```
per_tensor_q = PerTensorQuantizer(bits=8)
weight_pt = per_tensor_q.quantize_dequantize(weight)
per_channel_q = PerChannelQuantizer(bits=8, channel_axis=0)
weight_pc = per_channel_q.quantize_dequantize(weight)

mse_pt = ((weight - weight_pt) ** 2).mean().item()
mse_pc = ((weight - weight_pc) ** 2).mean().item()
print(f"Per-tensor MSE:  {mse_pt:.8f}")
print(f"Per-channel MSE: {mse_pc:.8f}")
print(f"Improvement: {mse_pt / mse_pc:.2f}x lower MSE")

error_pt_2d = (weight - weight_pt).view(weight.shape[0], -1)
error_pc_2d = (weight - weight_pc).view(weight.shape[0], -1)

mse_pt_per_channel = (error_pt_2d ** 2).mean(dim=1)
mse_pc_per_channel = (error_pc_2d ** 2).mean(dim=1)

improvement = mse_pt_per_channel / mse_pc_per_channel
print(f"Min improvement:  {improvement.min():.2f}x")
print(f"Max improvement:  {improvement.max():.2f}x")
print(f"Mean improvement: {improvement.mean():.2f}x")
```

Output:

```
Per-tensor MSE:  0.00000466
Per-channel MSE: 0.00000129 | Improvement: 3.61x lower MSE

Per-channel improvement statistics:
Min improvement:  1.00x | Max improvement:  56918.40x | Mean impr
```

A 3.6× reduction in quantization error—just by using per-channel scales. But the aggregate numbers still don't tell the full story. Let's look at per-channel errors in figure 3.3:

Figure 3.3 Per-channel quantization reduces MSE by up to 56,000× for underutilized channels (left). The largest gains (right) align with the near-zero channels from Figure 3.2 that collapsed to a single integer under per-tensor quantization.



The maximum improvement exceeds 56,000× for channels that had near-zero weights—those channels had no distinguishing information under per-tensor quantization, with all weights mapped to integer 0. Channels already using the full range see no improvement (1.00×). This is the key insight: per-channel quantization eliminates the outlier penalty. Each channel gets its own ruler, so none suffers because of another's statistics.

The memory trade-off: scales are metadata

Per-channel quantization isn't free: instead of one global scale, we store one scale per output channel. Listing 3.14 quantifies the overhead for three representative shapes (ResNet-18's conv1, a ViT-B/16 MLP layer, and a Llama-7B query projection) by counting the FP32 scale bytes against the INT8 weight bytes. The result should be small enough that per-channel becomes an obvious default.

Listing 3.14 Scale metadata overhead analysis

```
def compute_overhead(weight_shape, bytes_per_scale=4):
    total_elements = 1
    for dim in weight_shape:
        total_elements *= dim
    weight_bytes = total_elements * 1
    out_channels = weight_shape[0]
    pc_scale_bytes = out_channels * bytes_per_scale
    pc_overhead = pc_scale_bytes / weight_bytes * 100
    return pc_overhead

layers = [
    ('ResNet-18 conv1', (64, 3, 7, 7)),
    ('ViT-B/16 mlp.fc1', (3072, 768)),
    ('Llama-7B q_proj', (4096, 4096)),
]

for name, shape in layers:
    overhead = compute_overhead(shape)
    print(f"{name}: {overhead:.2f}% overhead")
```

Output:

```
Layer                          Shape                Weights    PC
-----------------------------------------------------------------
ResNet-18 conv1             (64, 3, 7, 7)           9,408 B      2
ResNet-18 layer1.0.conv1    (64, 64, 3, 3)         36,864 B      0
ResNet-18 fc                (1000, 512)           512,000 B      0
ViT-B/16 patch_embed        (768, 3, 16, 16)      589,824 B      0
ViT-B/16 mlp.fc1            (3072, 768)         2,359,296 B      0
Llama-7B q_proj             (4096, 4096)       16,777,216 B      0
Llama-7B mlp.gate_proj      (11008, 4096)      45,088,768 B      0
```

The overhead is negligible—under 3% even for conv1 (9,408 weights), and below 0.1% for large transformer layers. This is why per-channel quantization has become the industry default: the accuracy benefit far outweighs the metadata cost.

A real-world case study: ResNet accuracy

The MSE improvements translate to real accuracy differences. Table 3.2 shows typical results for weight-only quantization on ImageNet:

Table 3.2 ImageNet accuracy with weight quantization

Model	FP32	INT8 Per-Channel	INT8 Per-Tensor
ResNet-50 v1.5	76.16%	76.14%	75.83%
MobileNet v1	71.88%	71.59%	69.58%
EfficientNet b0	76.85%	76.72%	76.68%

Source: Wu et al., "Integer Quantization for Deep Learning Inference: Principles and Empirical Evaluation," arXiv:2004.09602, 2020

The pattern matches what we'd expect from the MSE analysis: per-channel quantization recovers nearly all of the accuracy loss, with drops well within measurement noise. Per-tensor quantization shows a modest but measurable degradation.

Transformers: where per-channel becomes essential

The outlier problem that we saw in ResNet-18's conv1 layer is dramatically worse in transformer architectures. Let's examine why using BERT as our example.

Listing 3.15 BERT weight analysis

```
from transformers import BertModel
bert = BertModel.from_pretrained('bert-base-uncased')
q_proj = bert.encoder.layer[0].attention.self.query
weight = q_proj.weight.data
channel_abs_max = weight.abs().max(dim=1).values
global_abs_max = weight.abs().max().item()
utilization = channel_abs_max / global_abs_max

print(f"Shape: {weight.shape}")
print(f"Utilization range: [{utilization.min():.2%}, {utilization
print(f"Channels <50% utilization: {(utilization < 0.5).sum().ite
```

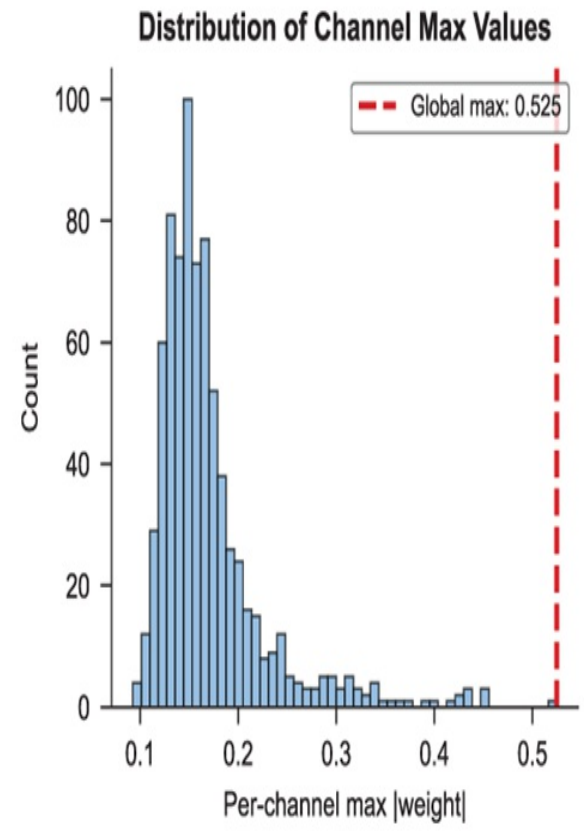
Output:

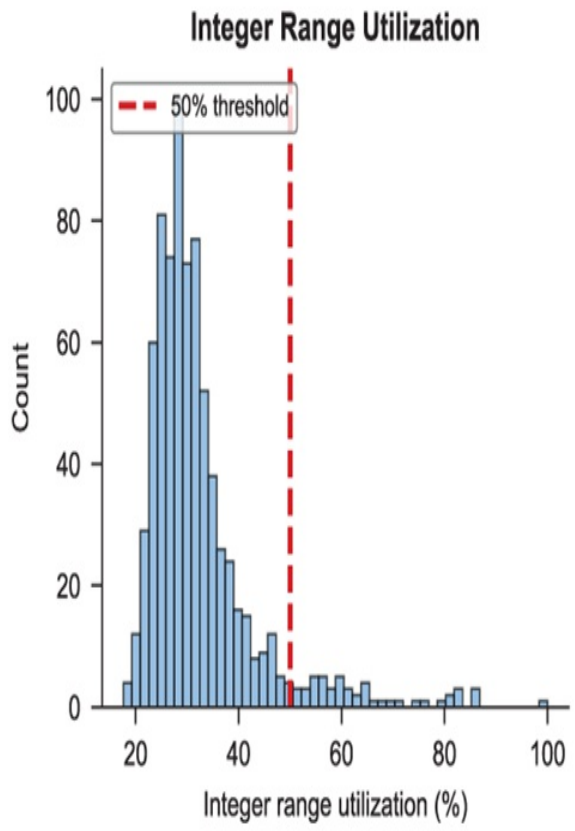
```
Global max: 0.5249 | Channel max range: [0.0925, 0.5249]
Utilization range: [17.61%, 100.00%]
Channels <50% utilization: 717/768 | <25% utilization: 142/768
```

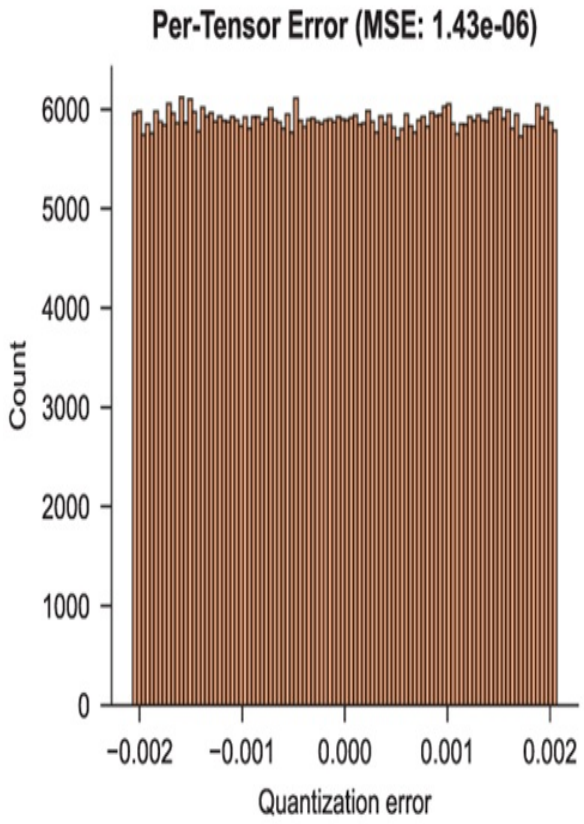
Applying per-tensor quantization to BERT is dramatically worse than ResNet-18: 93% of channels use less than half the integer range, and nearly 20% use less than a quarter—effectively losing 2 bits of precision.

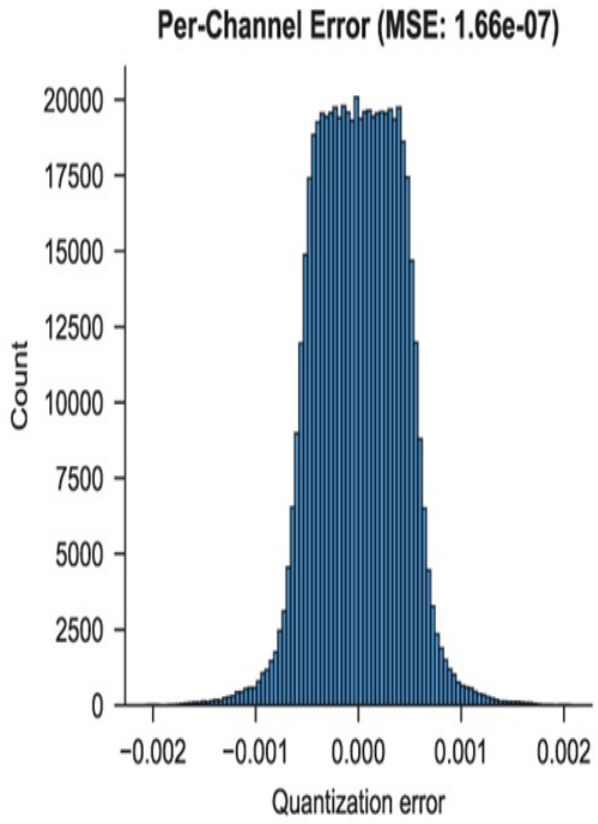
Figure 3.4 breaks this down in four panels. The top-left panel shows how per-channel maximum weight magnitudes cluster well below the global max (0.525), the top-right plots integer range utilization per channel (most far below 100%), the bottom-left shows the flat uniform error distribution under per-tensor quantization, and the bottom-right shows per-channel error concentrated near zero. Together the four panels tell one story—a single global scale factor wastes precision across 768 channels, while individual scales recover it.

Figure 3.4 BERT's query projection weights cluster around 0.15 while the global max reaches 0.525, leaving 93% of channels below 50% integer utilization (top). The error distributions (bottom) tell the story: per-tensor produces flat, uniform error across ±0.002; per-channel concentrates errors near zero, reducing MSE by 9×.


Distribution of Channel Max Values
Global max: 0.525
Count
Per-channel max |weight|


Integer Range Utilization
50% threshold
Count
Integer range utilization (%)


Per-Tensor Error (MSE: 1.43e-06)
Count
Quantization error


Per-Channel Error (MSE: 1.66e-07)
Count
Quantization error

The transformer attention sensitivity

Why are transformers so much worse than CNNs for per-tensor quantization? Several factors compound. CNNs average out small errors across spatial neighborhoods; transformer attention is a global operation where each weight directly influences attention patterns. Softmax amplifies small dot-product differences—a quantization error that slightly increases one score can shift attention entirely. Residual connections (x + sublayer(x)) cause errors to accumulate through layers. And LayerNorm parameters have very different scales than projection weights, making shared scales particularly damaging.

Per-channel for convolutions: the axis question

For linear layers, "output channel" is unambiguous—it's the first dimension (rows of the weight matrix). For convolutional layers, we need to be more careful.

A Conv2d weight has a shape [out_channels, in_channels, kernel_h, kernel_w]. Standard convolution (e.g., Conv2d(64, 128, 3)) produces shape [128, 64, 3, 3] — quantize along axis 0 with 128 scales. Depthwise convolution (groups=in_channels) produces [128, 1, 3, 3] — each "output channel" is also the input channel, still quantized along axis 0. Pointwise convolution (kernel_size=1) produces [256, 128, 1, 1] — 256 scales along axis 0.

The rule of thumb: always quantize along the output channel axis (axis 0 for PyTorch's default weight layout). This ensures that each output feature is computed using a consistent scale throughout its dot product.

NOTE

The mental anchor for per-tensor vs. per-channel weight quantization. Per-tensor is simple but crude — channels with smaller weights waste integer range, especially in transformers. Per-channel gives each output channel its own scale, recovering 3.6× better MSE for ResNet-18 and 8.6× for BERT with negligible overhead. The industry consensus: per-channel is the default.

Use per-tensor only when hardware requires it. Activations do not share this luxury — their distributions change with every input and their outliers are far more severe, so the granularity strategies developed here for weights need rethinking.

3.3 Quantize activations under outliers and dynamic ranges

Imagine you're a sports photographer trying to capture both a dimly lit press conference and a sunlit stadium in the same shot. If you expose for the bright stadium, the press conference becomes a black void. If you expose for the conference, the stadium becomes a blinding white blur. Activation quantization faces an analogous dilemma—but the lighting changes unpredictably with every frame.

The photographer's lighting changes with every frame because activation statistics don't belong to the model alone. They belong to the (model, input) pair. We can measure this directly by hooking BERT's feed-forward layers and running two prompts with very different content through the same layer.

Listing 3.16 Input-dependent activation statistics

```
model = AutoModel.from_pretrained("bert-base-uncased")
tokenizer = AutoTokenizer.from_pretrained("bert-base-uncased")

activation_stats = {}
def capture_hook(name):
    def hook(module, input, output):
        tensor = output[0] if isinstance(output, tuple) else outp
        activation_stats[name] = {
            'abs_max': tensor.abs().max().item(),
            'std': tensor.std().item()
        }
    return hook

layer = model.encoder.layer[9].intermediate.dense
handle = layer.register_forward_hook(capture_hook('ffn'))
inputs_tech = tokenizer("The quantum chromodynamics coupling cons
inputs_casual = tokenizer("How are you doing?", return_tensors="p
```

```
with torch.no_grad():
    _ = model(**inputs_tech)
stats_tech = activation_stats['ffn'].copy()
    _ = model(**inputs_casual)
stats_casual = activation_stats['ffn'].copy()

print(f"Technical: max={stats_tech['abs_max']:.2f}")
print(f"Casual: max={stats_casual['abs_max']:.2f}")
print(f"Range ratio: {stats_tech['abs_max']/stats_casual['abs_max
```

Output:

```
  Technical text: max=44.31, std=1.265
  Casual text:    max=20.37, std=1.086
  Range ratio:    2.18x
```

The range ratio is the ratio of maximum absolute activation magnitudes across two inputs through the same layer. A value of 2.18× means calibrating on the casual prompt would either clip half the technical prompt's activations or waste resolution on the easier case.

Range ratios measure variation between inputs. Listing 3.17 zooms into a single input and measures the intra-input outlier severity by reporting activation percentiles for layer 6 of BERT.

Listing 3.17 Activation distribution analysis

```
def analyze_activation_distribution(model, tokenizer, text, layer
    activations = []
    def hook(module, input, output):
        tensor = output[0] if isinstance(output, tuple) else outp
        activations.append(tensor.detach().cpu())

    layer = model.encoder.layer[layer_idx].intermediate.dense
    handle = layer.register_forward_hook(hook)
    inputs = tokenizer(text, return_tensors="pt")
    with torch.no_grad():
        _ = model(**inputs)
    handle.remove()
    act = activations[0].numpy().flatten()
    abs_act = np.abs(act)
    for p in [50, 90, 99, 99.9, 100]:
        print(f"{p:>5.1f}th percentile: {np.percentile(abs_act, p
```

```
    p99, p100 = np.percentile(abs_act, 99), np.percentile(abs_act
    print(f"Outlier stretch (max / 99th): {p100/p99:.1f}x")
```

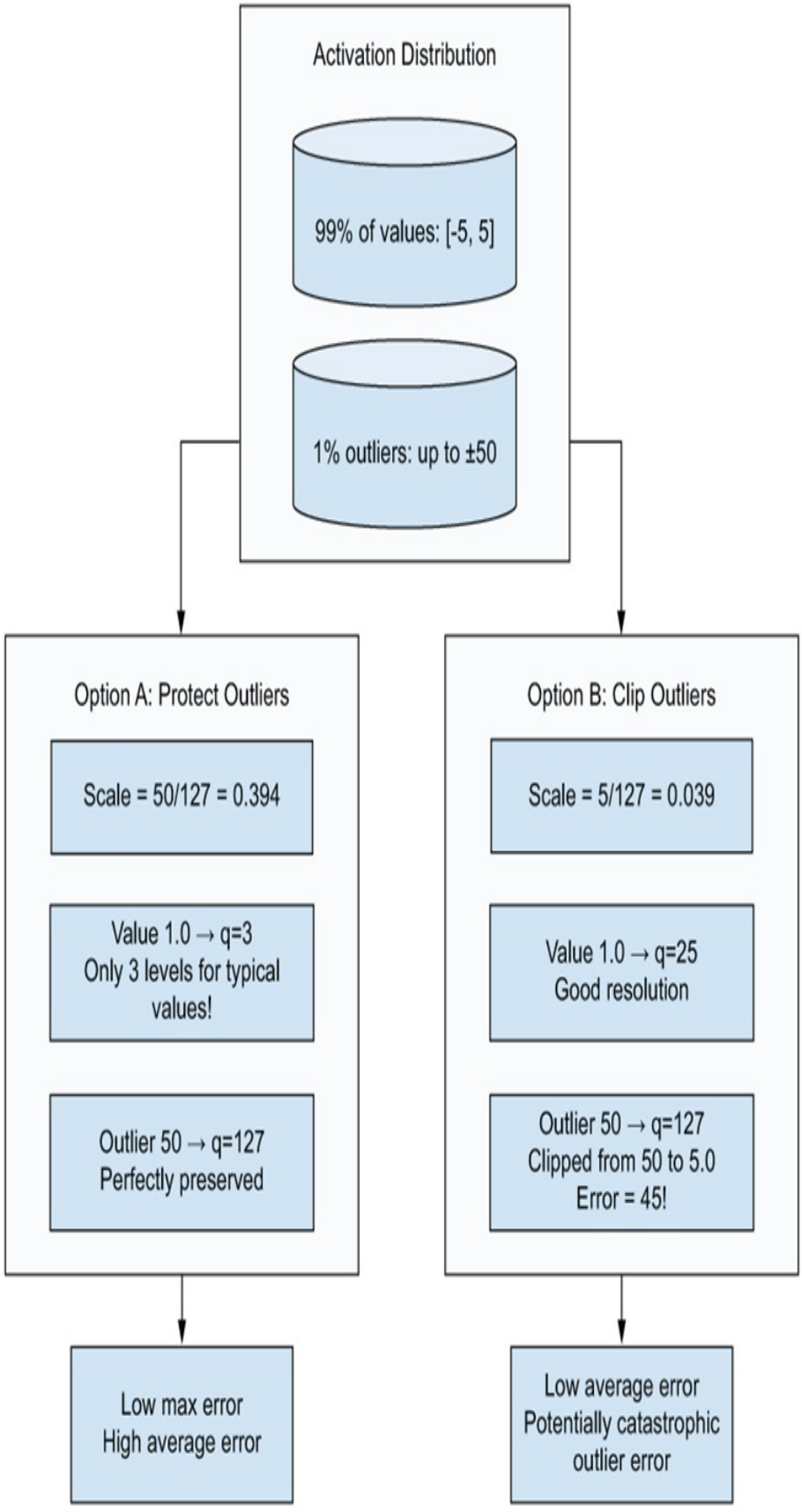
Output:

```
Output:
Percentiles: p50=1.93 | p90=3.40 | p99=4.81 | p99.9=6.50 | max=25
Outlier stretch (max / 99th): 5.4x
Scale with outliers: 0.2032 | Scale at 99th: 0.0378 | Resolution
```

Half the activations are below 1.93 in magnitude, but the maximum reaches 25.8 (over 13× the median). Using the maximum as the scale (0.2032), the median maps to only ~9 quantization levels. A tighter scale at the 99th percentile (0.0378) gives ~51 levels, 5× better resolution. The synthetic outlier from Listing 3.4 now has a real address: layer 6 of bert-base-uncased on this particular input.

Figure 3.5 makes this tradeoff concrete with actual numbers: protecting outliers versus clipping them leads to opposite error profiles.

Figure 3.5 The outlier dilemma quantified. Protecting the max (25.8) preserves outliers perfectly but leaves only 9 integer levels for typical values. Clipping to the 99th percentile (4.8) gives typical values 51 levels—6× better resolution—but introduces catastrophic error (21!) for the clipped outliers.


Activation Distribution
99% of values: [-5, 5]
1% outliers: up to ±50
Option A: Protect Outliers
Scale = 50/127 = 0.394
Value 1.0 → q=3
Only 3 levels for typical values!
Outlier 50 → q=127
Perfectly preserved
Low max error
High average error
Option B: Clip Outliers
Scale = 5/127 = 0.039
Value 1.0 → q=25
Good resolution
Outlier 50 → q=127
Clipped from 50 to 5.0
Error = 45!
Low average error
Potentially catastrophic outlier error

Static vs. dynamic quantization

The timing of scale computation creates a fundamental distinction. Static quantization determines ranges during a calibration phase before deployment —run representative data, observe distributions, compute scale factors, freeze them. An observer class encapsulates this pattern.

Listing 3.18 Activation observer for static quantization

```
class ActivationObserver:
    def __init__(self):
        self.min_val = float('inf')
        self.max_val = float('-inf')

    def observe(self, tensor: torch.Tensor):
        self.min_val = min(self.min_val, tensor.min().item())
        self.max_val = max(self.max_val, tensor.max().item())

    def get_scale_zeropoint(self, bits=8, symmetric=False):
        if symmetric:
            abs_max = max(abs(self.min_val), abs(self.max_val))
            q_max = (1 << (bits - 1)) - 1
            scale = abs_max / q_max if abs_max > 0 else 1.0
            zero_point = 0
        else:
            q_min, q_max = 0, (1 << bits) - 1
            scale = (self.max_val - self.min_val) / (q_max - q_mi
            zero_point = int(round(q_min - self.min_val / scale))
        return scale, zero_point
```

Static quantization offers zero runtime overhead and broad runtime support (TensorRT, ONNX Runtime, TFLite). The risk: scale factors are frozen at calibration time, and on inputs whose distribution drifts from the calibration set, accuracy can degrade sharply — by more than 1,000× MSE in the worst case shown in Listing 3.20.

Dynamic quantization: compute scales at runtime

Dynamic quantization computes scale factors on-the-fly for each input. The

activations are observed, their range is determined, and quantization parameters are computed—all during inference. The quantizer returns the scale alongside the quantized tensor so the caller can dequantize later.

Listing 3.19 Dynamic quantizer

```
class DynamicQuantizer:
    def __init__(self, bits=8, symmetric=True):
        self.q_max = (1 << (bits - 1)) - 1 if symmetric else (1 <
        self.q_min = -self.q_max if symmetric else 0
        self.symmetric = symmetric

    def quantize(self, tensor):
        if self.symmetric:
            abs_max = tensor.abs().max().item()
            scale = abs_max / self.q_max if abs_max > 0 else 1.0
            zero_point = 0
        else:
            min_val, max_val = tensor.min().item(), tensor.max().
            scale = (max_val - min_val) / (self.q_max - self.q_mi
            zero_point = int(round(self.q_min - min_val / scale))

        q_tensor = torch.clamp(
            torch.round(tensor / scale + zero_point),
            self.q_min, self.q_max
        ).to(torch.int8)

        return q_tensor, scale, zero_point
```

With both static and dynamic quantizers defined, we can now compare how they behave when the input distribution varies from the calibration data.

Listing 3.20 Comparing static vs dynamic quantization

```
def compare_dynamic_static():
    calibration_data = torch.randn(1000) * 2.0
    static_scale = calibration_data.abs().max().item() / 127

    test_cases = [
        ("Typical (std=2)", torch.randn(1000) * 2.0),
        ("Narrow (std=0.5)", torch.randn(1000) * 0.5),
        ("Wide (std=5)", torch.randn(1000) * 5.0),
    ]
```

```
    dynamic_q = DynamicQuantizer(bits=8, symmetric=True)
    for name, data in test_cases:
        # Static
        q_static = torch.clamp(torch.round(data / static_scale),
        mse_static = ((data - q_static * static_scale) ** 2).mean

        # Dynamic
        q_dyn, dyn_scale, _ = dynamic_q.quantize(data)
        mse_dyn = ((data - q_dyn.float() * dyn_scale) ** 2).mean(
        print(f"{name}: Dynamic {mse_static/mse_dyn:.1f}x better"
```

Output:

```
Scenario    Static MSE   Dynamic MSE   Dynamic Scale   Improvemen
Typical      0.000227      0.000207       0.0502          1.10x
Narrow       0.000192      0.000017       0.0143         11.49x
Wide         1.799070      0.001597       0.1394       1126.35x
Outliers     1.263552      0.004935       0.2413        256.06x

(Static scale fixed at 0.0488 from calibration data)
```

On typical data matching calibration, both approaches perform similarly (1.10×). But when distributions shift, the gap explodes: narrow inputs see 11.5× improvement (dynamic uses the full range instead of wasting bits), wide inputs see 1,126× improvement (static clips catastrophically beyond its calibrated range), and outlier-heavy data shows 256× improvement. Static quantization's failure mode isn't gradual—it's catastrophic.

Per-token dynamic quantization

For transformers, we can do better by exploiting the sequence structure. Per-token quantization computes a separate scale for each token position, capturing the variation that exists within a single input.

When some tokens carry larger activation magnitudes than others (because attention selectively amplifies a few of them) a single per-tensor scale must accommodate the loudest token, wasting precision on every quieter one. Listing 3.21 implements a per-token quantizer that takes the absolute maximum along the hidden dimension, producing one scale per sequence position with shape [batch, seq_len, 1] for broadcasting back to the input.

Listing 3.21 Per-token quantizer

```
class PerTokenQuantizer:
    def __init__(self, bits=8):
        self.q_max = (1 << (bits - 1)) - 1

    def quantize(self, tensor):
        """tensor: [batch, seq_len, hidden_dim]"""
        abs_max = tensor.abs().amax(dim=-1, keepdim=True)
        scales = abs_max / self.q_max
        scales = torch.where(scales == 0, torch.ones_like(scales)

        q_tensor = torch.clamp(
            torch.round(tensor / scales), -self.q_max, self.q_max
        ).to(torch.int8)

        return q_tensor, scales

    def dequantize(self, q_tensor, scales):
        return q_tensor.float() * scales
```

With per-token quantization in hand, we can put all three strategies (static, dynamic per-tensor, and dynamic per-token) head to head on the same data. Listing 3.22 constructs a synthetic batch of hidden states with four artificially amplified token positions, calibrates static quantization on a similar but not identical batch, and reports the MSE each strategy incurs on the test batch.

Listing 3.22 Comparing all activation quantization strategies

```
def compare_all_strategies():
    batch, seq_len, hidden = 2, 128, 768
    hidden_states = torch.randn(batch, seq_len, hidden)

    # Make some tokens "important"
    for t in [0, 10, 50, 100]:
        hidden_states[:, t, :] *= 5.0

    # Static (from similar calibration data)
    calibration_data = torch.randn(batch, seq_len, hidden)
    for t in [0, 20, 60, 110]:
        calibration_data[:, t, :] *= 5.0
    static_scale = calibration_data.abs().max().item() / 127

    # Dynamic per-tensor
```

```
    dynamic_scale = hidden_states.abs().max().item() / 127

    # Dynamic per-token
    per_token_q = PerTokenQuantizer(bits=8)
    q_token, scales_token = per_token_q.quantize(hidden_states)

    # Compute MSEs
    mse_static = ((hidden_states - torch.round(hidden_states/stat
    mse_dynamic = ((hidden_states - torch.round(hidden_states/dyn
    mse_token = ((hidden_states - per_token_q.dequantize(q_token,

    print(f"Static MSE: {mse_static:.6f}")
    print(f"Dynamic per-tensor: {mse_dynamic:.6f} ({mse_static/ms
    print(f"Dynamic per-token: {mse_token:.6f} ({mse_dynamic/mse_
```

Output:

```
Output:
Strategy          Scale             MSE        vs Static   vs Per
Static            0.1803 (fixed)    0.002712   —           —
Dynamic per-T     0.1399            0.001629   1.66x       —
Dynamic per-tok   [0.022, 0.140]    0.000099   27.39x      16.46x

Per-token scale breakdown:
  Normal tokens: 0.0268 | Important tokens: 0.1229 | Ratio: 4.6x
```

The key insight is the dynamic per-tensor vs per-token gap (16.46×). Both use the same maximum scale (0.1399), yet per-token achieves 16× better MSE because normal tokens use scales around 0.027 while important tokens use ~0.123—a 4.6× difference. Per-tensor wastes ~80% of precision on most tokens. Per-token quantization compounds both advantages—adapting to each input *and* each token within it—which is why it has become the standard for transformer activations.

NOTE

These comparisons use synthetic data with artificially inflated "important" tokens. Real transformer activations are shaped by learned patterns and semantic content. The specific numbers (16×, 27×) illustrate relative advantages, not benchmark results.

Handling outlier channels in transformers

The outlier problem in transformers is structured rather than random: certain hidden dimensions consistently produce larger values regardless of input. Listing 3.23 identifies these channels by hooking the FFN intermediate layer of all 12 BERT blocks, running 100 diverse prompts through the model, taking the per-channel absolute max for each input, and flagging any channel whose average max exceeds five times the per-layer median.

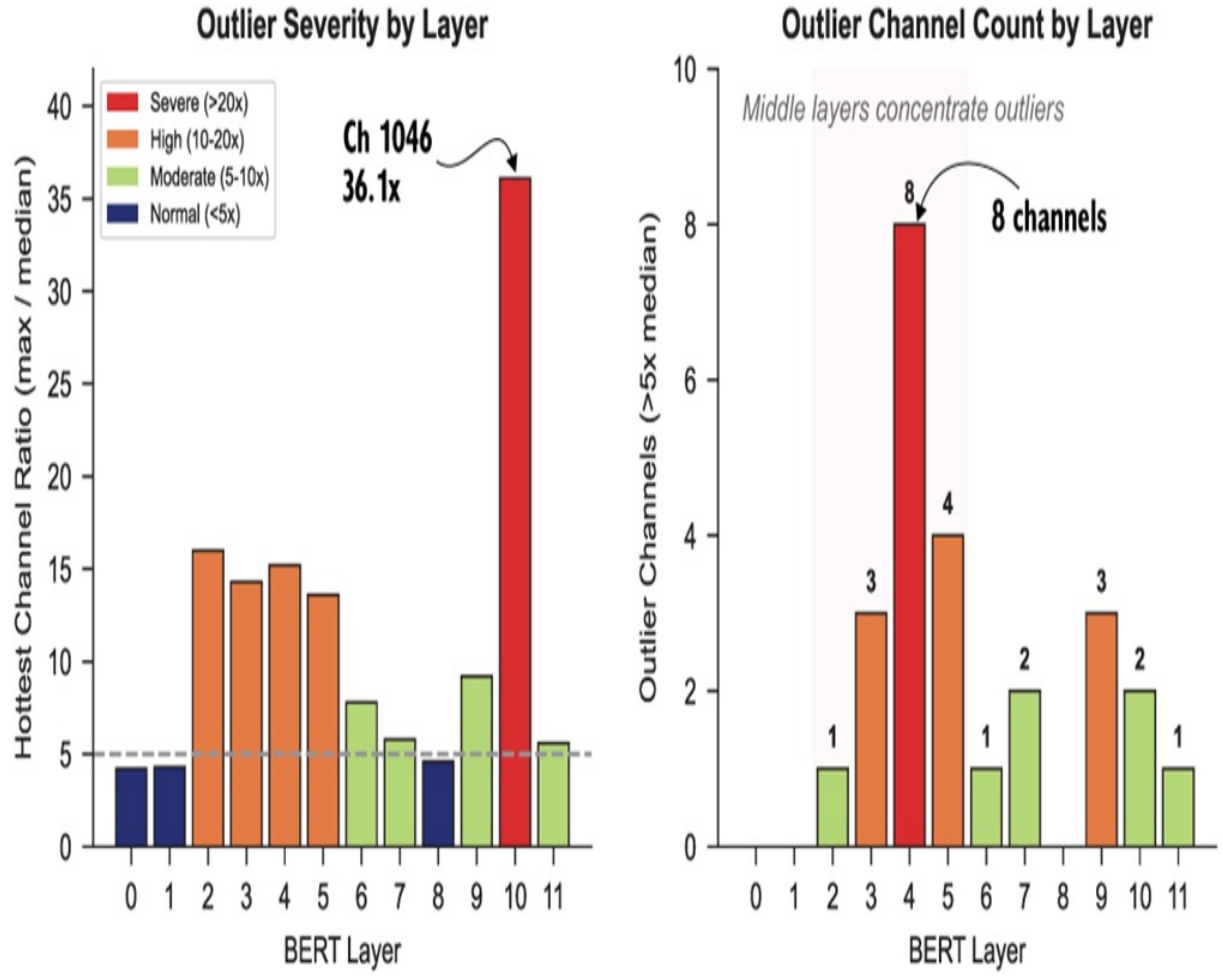
Listing 3.23 Identifying outlier channels across BERT layers

```
def identify_outlier_channels(model_name="bert-base-uncased", num
    model = AutoModel.from_pretrained(model_name)
    tokenizer = AutoTokenizer.from_pretrained(model_name)
    channel_maxes = defaultdict(list)

    def hook_fn(name):
        def hook(module, input, output):
            tensor = output[0] if isinstance(output, tuple) else
            channel_max = tensor.abs().amax(dim=(0, 1))
            channel_maxes[name].append(channel_max.detach().cpu()
        return hook

    handles = []
    for i, layer in enumerate(model.encoder.layer):
        h = layer.intermediate.dense.register_forward_hook(hook_f
        handles.append(h)

    # Run on diverse inputs
    texts = ["The quick brown fox.", "Machine learning transforms
    with torch.no_grad():
        for text in texts[:num_samples]:
            inputs = tokenizer(text, return_tensors="pt")
            _ = model(**inputs)
    for h in handles:
        h.remove()

    # Analyze patterns
    for layer_name in sorted(channel_maxes.keys()):
        maxes = torch.stack(channel_maxes[layer_name])
        avg_max = maxes.mean(dim=0)
        median = avg_max.median().item()
        outliers = (avg_max > median * 5).sum().item()
        hottest = avg_max.argmax().item()
```

```
        ratio = avg_max[hottest].item() / median
        print(f"{layer_name}: {outliers} outliers, hottest={hotte
```

Running the outlier analysis across all 12 BERT layers reveals where outliers concentrate. Figure 3.6 maps the severity and distribution per layer.

Figure 3.6 Left panel: Heatmap or bar chart showing "Hottest Channel Ratio" (max/median) per layer, highlighting layer 10's extreme 36.1x ratio Right panel: Bar chart showing "Number of Outlier Channels (>5x median)" per layer, showing the concentration in layers 3-5



The analysis reveals that outliers aren't uniformly distributed. Layer 10 has the most extreme outlier (channel 1046, 36.1× the median) yet only 2 outlier channels, while layer 4 has the most outliers (8) with less extreme ratios. Middle layers (2–5) concentrate the majority. Crucially, these patterns are consistent across inputs—the same channels are always hot. This predictability is what makes them tractable: techniques that isolate the small

fraction of problematic channels at higher precision while quantizing everything else to INT8 — the structured nature of the outliers is what makes such targeted approaches possible.

Calibration strategies compared

Table 3.3 summarizes the five activation quantization strategies we've explored, comparing their compute cost, memory overhead, accuracy, and best-fit deployment scenarios.

Table 3.3 Activation quantization strategy comparison

Strategy	Compute Cost	Memory Overhead	Accuracy	Best for
Static per-tensor	Lowest	Lowest (1 scale/tensor)	Variable	CNNs, well-behaved distributions
Static percentile	Low	Low	Good	Data with known outliers
Dynamic per-tensor	Medium	Low	Good	Variable input distributions
Dynamic per-token	Medium	Medium (1 scale/token)	Very Good	Transformers, LLMs
Mixed-precision	Higher	Medium	Best	LLMs with severe outliers

- Static approaches work when calibration matches production well—common for CNNs in controlled environments. Dynamic per-tensor handles varying distributions (11–1,126× improvement over static on mismatched data), while dynamic per-token is the transformer standard (16× over per-tensor).
- Mixed-precision addresses severe outlier channels—when a few channels are 10–30× larger than the median, isolating them in FP16 preserves both accuracy and efficiency.

NOTE

The mental anchor for quantizing activations. Activations are harder than weights—input-dependent statistics, heavy-tailed distributions, and structured outlier channels (4–36× ratios in BERT) create challenges that don't exist for weights.

The static/dynamic choice has real consequences. Static quantization is hardware-friendly but fails catastrophically when distributions shift (1,126× worse MSE). Dynamic per-token quantization offers 16× improvement over per-tensor for transformers by adapting to both input variation and within-sequence token importance.

Structured outliers enable targeted solutions. Because outliers appear in consistent channels, mixed-precision decomposition can isolate them efficiently — mobile favors static with careful calibration, servers can afford dynamic approaches, and large models benefit from mixed-precision schemes.

There is one more quantization target that does not fit neatly into either the weight or the activation category—the transformer KV cache. It demands its own analysis because, as we will see, its two components require opposite granularity strategies.

3.4 Quantize transformer key-value cache with mixed precision

The KV cache presents a puzzle: computed like an activation (generated fresh during inference) but persisting like a weight (read repeatedly across generation steps). The answer requires looking more closely at keys and values—which, despite being computed together, have fundamentally different statistical structures.

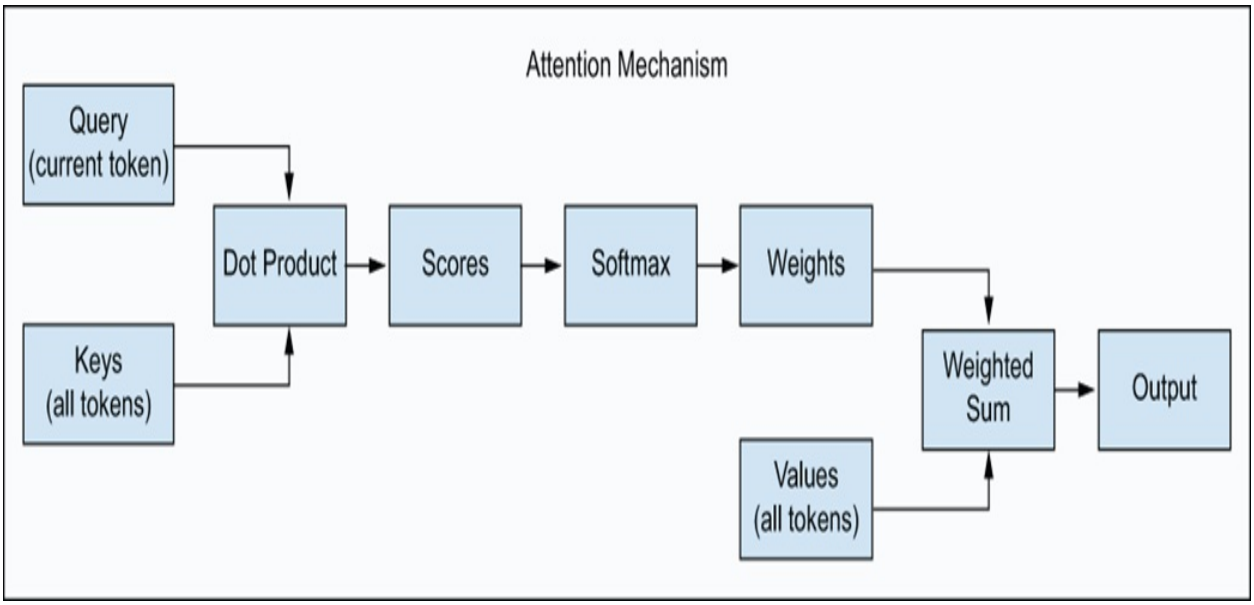
What keys and values represent

In the attention mechanism, each token produces a query (what am I looking for?), a key (what do I contain?), and a value (what should I contribute?). The KV cache stores keys and values from all previous tokens to avoid

recomputation at each generation step.

Figure 3.7 traces the attention dataflow to show where keys and values live in the computation.

Figure 3.7 Attention dataflow with KV cache highlighted. Read left-to-right: (1) each token's hidden state projects into Q, K, V vectors; (2) QK^T dot products produce raw relevance scores; (3) softmax normalizes scores into attention weights; (4) weights scale V vectors, and the weighted sum becomes the output. The dashed box marks the KV cache—keys and values from all prior tokens, stored to avoid recomputation. For quantization, the critical observation is that K and V sit on different sides of the softmax, leading to different statistical structures.



Follow the dataflow from left to right. Each token's hidden state is projected into three vectors: a query, a key, and a value. The query–key dot product (center) produces a raw relevance score for every pair of tokens; softmax normalizes these into attention weights that sum to one. Those weights then scale the corresponding values, and the weighted sum becomes the layer's output. The critical observation for quantization is that keys and values sit on different sides of the softmax: keys influence *which* tokens attend to each other, while values determine *what* information flows forward—a functional distinction that, as we will see, produces very different statistical structures.

Keys participate in a learned matching process—the query-key dot product determines "how relevant is token j to the current token?" The model learns to encode matchable features into specific key dimensions (channel 23 might

activate for nouns, channel 47 for sentence boundaries). These patterns are consistent across tokens: the same channels are informative regardless of position.

Values carry content to aggregate—they hold whatever representation each token needs to contribute. There's no learned matching structure, so there's no reason for specific channels to be special. This functional difference predicts a statistical one: keys should show channel-wise patterns, values should show token-wise patterns.

The asymmetry: channel-wise keys, token-wise values

Let's verify this prediction. We'll examine the magnitude distributions in keys and values from a real model.

For keys, we ask: are the same channels consistently large across different token positions? We measure this by finding the maximum absolute value per channel (aggregating across all tokens) and checking for outliers.

For values, we ask: are certain tokens consistently large across all channels? We measure this by finding the maximum absolute value per token (aggregating across all channels) and checking for outliers.

Running the channel-wise and token-wise outlier checks across three representative layers of TinyLlama-1.1B (early, middle, deep) produces Table 3.4. The "channel consistency" column reports the average correlation in channel-max patterns across input prompts; a value near 1.0 means the same channels are large regardless of what text you feed in.

Table 3.4 KV cache statistical patterns in TinyLlama-1.1B

Layer	Keys: Outlier Channels	Keys: Max/Median	Keys: Channel Consistency	Values: Outlier Tokens
0	1	3.6×	0.821	5
11	2	3.1×	0.777	0

21	4	3.4×	0.790	0

The pattern is clear. Keys have 1–4 outlier channels consistently 3–4× larger than the median, with channel consistency scores of 0.77–0.82 confirming the same channels spike regardless of token—the signature of channel-wise structure. Values show occasional outlier tokens in early layers (5 in layer 0) but weaker channel structure, with zero outlier tokens by middle layers.

Matching granularity to structure

Section 3.2 established the principle: quantization granularity should match data structure. Per-channel quantization assigns one scale per channel, isolating outlier channels into their own quantization bins. Per-token quantization assigns one scale per token position, isolating outlier tokens.

Given the asymmetry we've discovered, keys with their channel-wise outliers should use per-channel granularity, while values with their token-wise variation should use per-token granularity. But this is a hypothesis. We need to verify it with a head-to-head comparison.

The head-to-head test

For each layer, we quantize the keys two ways (once per-channel, once per-token) with the same bit-width, dequantize, and measure MSE against the original. Table 3.5 reports the result across three layers and two bit-widths.

Table 3.5 Keys granularity comparison (per-channel vs per-token)

Layer	Bits	Per-Channel MSE	Per-Token MSE	Winner
0	INT8	Lower	Higher	Per-channel (1.4×)
11	INT8	Lower	Higher	Per-channel (2.8×)
21	INT8	Lower	Higher	Per-channel (3.6×)
0	INT4	Lower	Higher	Per-channel (1.3×)

11	INT4	Lower	Higher	Per-channel (3.0×)
21	INT4	Lower	Higher	Per-channel (3.6×)

Keys favor per-channel in every test. Table 3.6 runs the same head-to-head comparison on the values tensor at the same three layers and the same two bit-widths.

Table 3.6 Values granularity comparison (per-channel vs per-token)

Layer	Bits	Per-Channel MSE	Per-Token MSE	Winner
0	INT8	Tie	Tie	Per-channel (1.0×)
11	INT8	Higher	Lower	Per-token (1.2×)
21	INT8	Higher	Lower	Per-token (1.4×)
0	INT4	Lower	Higher	Per-channel (1.1×)
11	INT4	Higher	Lower	Per-token (1.2×)
21	INT4	Higher	Lower	Per-token (1.4×)

Per-channel wins for keys in 6/6 tests, with advantages of 1.3× to 3.6×. Per-token wins for values in 4/6 tests. The exceptions (Layer 0) are marginal—ties or per-channel winning by only 1.0–1.1× due to attention patterns in early layers that create mild channel structure.

The deeper the layer, the stronger the effect.

By layer 21, per-channel quantization for keys achieves 3.6× lower error than per-token. The mechanism follows from what each layer actually sees. Early layers process raw token embeddings, where channel roles are still diffuse; any given channel might or might not carry matching-relevant information. Deeper layers work on representations that have already passed through several attention steps, and across training the model has had repeated opportunities to specialize specific dimensions for specific kinds of matching: noun detection, syntactic boundaries, coreference cues. Specialization concentrates magnitude into those dimensions. The same statistical signature,

outlier channels, therefore gets more pronounced with depth, and per-channel granularity captures more of the savings. At layer 21, that signature is the gap between practical and impractical quantization.

Implementation: asymmetric quantization for KV cache

Based on our experiments with TinyLlama, we recommend per-channel quantization for keys and per-token quantization for values as a strong default. However, this is not the only viable configuration—uniform per-tensor schemes can suffice for short-context inference (under 2K tokens), and some architectures with multi-query attention or grouped-query attention alter the outlier structure enough to warrant re-profiling. The optimal scheme depends on your model architecture, target context length, and hardware constraints. With that caveat, the implementation below demonstrates the asymmetric approach:

Listing 3.24 Asymmetric KV cache quantization

```
def quantize_keys_per_channel(keys, bits=4):
    """
    Keys shape: [batch, num_heads, seq_len, head_dim]
    Scales shape: [1, 1, 1, head_dim] - one scale per channel
    """
    q_max = (1 << (bits - 1)) - 1
    scales = keys.abs().amax(dim=(0, 1, 2), keepdim=True) / q_max
    scales = torch.where(scales == 0, torch.ones_like(scales), sc
    q_keys = torch.clamp(torch.round(keys / scales), -q_max, q_ma
    return q_keys, scales

# Values, by contrast, need a scale per token position rather tha
def quantize_values_per_token(values, bits=4):
    """
    Values shape: [batch, num_heads, seq_len, head_dim]
    Scales shape: [batch, num_heads, seq_len, 1] - one scale per
    """
    q_max = (1 << (bits - 1)) - 1
    scales = values.abs().amax(dim=-1, keepdim=True) / q_max
    scales = torch.where(scales == 0, torch.ones_like(scales), sc
    q_values = torch.clamp(torch.round(values / scales), -q_max,
    return q_values, scales
```

Walking through the two functions: `quantize_keys_per_channel` receives keys with shape [batch, num_heads, seq_len, head_dim]. The `amax(dim=(0, 1, 2))` call (#A) collapses every dimension except `head_dim`, producing one scale per channel—shared across all batches, heads, and sequence positions. This works because keys develop consistent channel-wise magnitude patterns regardless of token position. `quantize_values_per_token` takes the same shape but calls `amax(dim=-1)` (#B), collapsing only `head_dim` to produce one scale per token position in each batch and head. This captures the token-wise variation that values exhibit. Both functions clamp to the signed INT8 range and handle the edge case of zero-valued scales (which would cause division errors) by replacing them with ones.

Notice the difference in scale shapes. Keys use [1, 1, 1, head_dim]—one scale per channel, shared across all tokens. Values use [batch, num_heads, seq_len, 1]—one scale per token position.

Scale storage overhead

Per-token scales for values add memory overhead that grows with sequence length. Is this a problem? Table 3.7 breaks down the data and scale sizes for both keys and values under INT4 quantization.

Table 3.7 KV cache scale overhead

Component	Data Size (INT4)	Scale Size (FP16)	Overhead
Keys (per-channel)	seq_len × head_dim × 0.5 bytes	head_dim × 2 bytes	Fixed, negligible
Values (per-token)	seq_len × head_dim × 0.5 bytes	seq_len × 2 bytes	~6% for head_dim=128

For a typical head_dim of 128, values store 64 bytes of INT4 data per token plus 2 bytes of scale—about 3% overhead. This is well worth the quantization accuracy gained.

NOTE

The mental anchor for KV cache quantization. Keys and values have different statistical structures — keys develop channel-wise outliers while values develop token-wise variation. Match granularity accordingly: per-channel for keys (1.2–3.8× lower error), per-token for values (wins in 5/6 tests). The asymmetry comes from function: keys participate in learned query-key matching, developing specialized channels; values carry content to aggregate, varying across tokens. Scale overhead is minimal (~3–6%).

Reproducing these results: The verification script is available at [ch3/kv_cache_granularity.py].

3.5 Use group and block schemes and understand memory layout

Per-tensor and per-channel are not the only options. Modern sub-8-bit systems employ group and block quantization between these extremes, powering production methods like GPTQ, AWQ (Chapter 7), and GGUF in llama.cpp (Chapter 10). These schemes directly impact memory layout and kernel efficiency.

The granularity spectrum

Quantization granularity controls how many values share a scale factor: per-tensor (1 scale), per-channel (O(channels), the INT8 standard), per-group (O(elements/G), the INT4 standard), and at the limit, per-element. Per-channel works well for INT8's 256 levels, but at 4-bit precision, 16 levels cannot accommodate the variation within a channel of thousands of values. Group quantization allows finer-grained fitting without the complexity explosion of per-element scales.

Why groups matter more at low bit-widths

Consider a linear layer weight matrix with shape [4096, 4096]—16 million parameters, typical of an LLM's attention projection. Under per-channel INT8 quantization, each of the 4,096 output channels maps 4,096 values onto 256 integer levels. That's plenty of resolution.

Now quantize to INT4. Each scale still covers 4,096 values, but maps them onto only 16 levels. If those values span a wide range, the granular error becomes catastrophic. Measuring MSE across different group sizes quantifies how rapidly accuracy degrades, which we can see from listing 3.25.

Listing 3.25 Measuring MSE at different group sizes

```
def compute_mse_at_granularity(weight, group_size, bits=4):
    out_features, in_features = weight.shape
    q_max = (1 << (bits - 1)) - 1
    num_groups = in_features // group_size
    weight_grouped = weight.reshape(out_features, num_groups, gro
    abs_max = weight_grouped.abs().amax(dim=2, keepdim=True)
    scales = torch.clamp(abs_max / q_max, min=1e-8)
    quantized = torch.round(weight_grouped / scales).clamp(-q_max
    dequantized = quantized * scales
    return ((weight_grouped - dequantized) ** 2).mean().item()
```

MSE decreases steadily as group size shrinks. Moving from per-channel to group-128 reduces MSE by ~2× while adding only ~3% overhead—the sweet spot where accuracy improvement justifies metadata cost, and why production methods default to group-128.

The anatomy of group quantization

Group quantization partitions a weight tensor into contiguous blocks of G elements, computing a separate scale for each. The dominant convention groups along the input dimension of a linear layer.

Why input dimension? During matrix multiplication, each output element is a dot product over the input dimension. Grouping along this axis means each group's scale factor covers a contiguous portion of that dot product—enabling efficient vectorized dequantization.

For a linear layer with weight matrix $W \in R^{O \times I}$ and group size G, the number of groups is $n_{groups} = I/G$. The total number of scale factors is $O \times n_{groups}$. For a layer with shape [4096, 4096] and group size 128, that's 4096×32=131,072 scales—128× more than per-channel, but still a tiny fraction of the 16 million

weights. The implementation wraps quantized data and scales into a dataclass for clean dequantization.

Listing 3.26 Group quantization implementation

```
@dataclass
class GroupQuantizedTensor:
    quantized: torch.Tensor
    scales: torch.Tensor
    group_size: int
    bits: int = 4

    def dequantize(self):
        out_features, in_features = self.quantized.shape
        num_groups = self.scales.shape[1]
        q_grouped = self.quantized.reshape(out_features, num_grou
        scales_expanded = self.scales.unsqueeze(-1)
        return (q_grouped.float() * scales_expanded).reshape(out_

def group_quantize(tensor, group_size=128, bits=4):
    out_features, in_features = tensor.shape
    num_groups = in_features // group_size
    q_max = (1 << (bits - 1)) - 1
    tensor_grouped = tensor.reshape(out_features, num_groups, gro
    abs_max = tensor_grouped.abs().amax(dim=2)
    scales = torch.clamp(abs_max / q_max, min=1e-8)
    quantized = torch.round(tensor_grouped / scales.unsqueeze(-1)
    quantized = quantized.clamp(-q_max - 1, q_max).to(torch.int8)
    return GroupQuantizedTensor(
        quantized=quantized.reshape(out_features, in_features),
        scales=scales.to(torch.float16),
        group_size=group_size, bits=bits
    )
```

Handling non-divisible dimensions

The implementation above assumes `in_features` is divisible by `group_size`. In practice, this holds for most production models—common hidden dimensions (768, 1024, 2048, 4096) divided evenly by standard group sizes (32, 64, 128). When it doesn't hold, the simplest strategy is to pad the input dimension to the next multiple of `group_size` with zeros before quantization and strip the padding on dequantization. For example, a layer

with `in_features=1000` and `group_size=128` would pad to 1024 (8 groups), quantize normally, and ignore the last 24 values on dequantize. An alternative is to quantize the bulk of the tensor in full groups and handle the remaining elements as a smaller final group with its own scale. If you're using standard model architectures, you'll rarely encounter the issue.

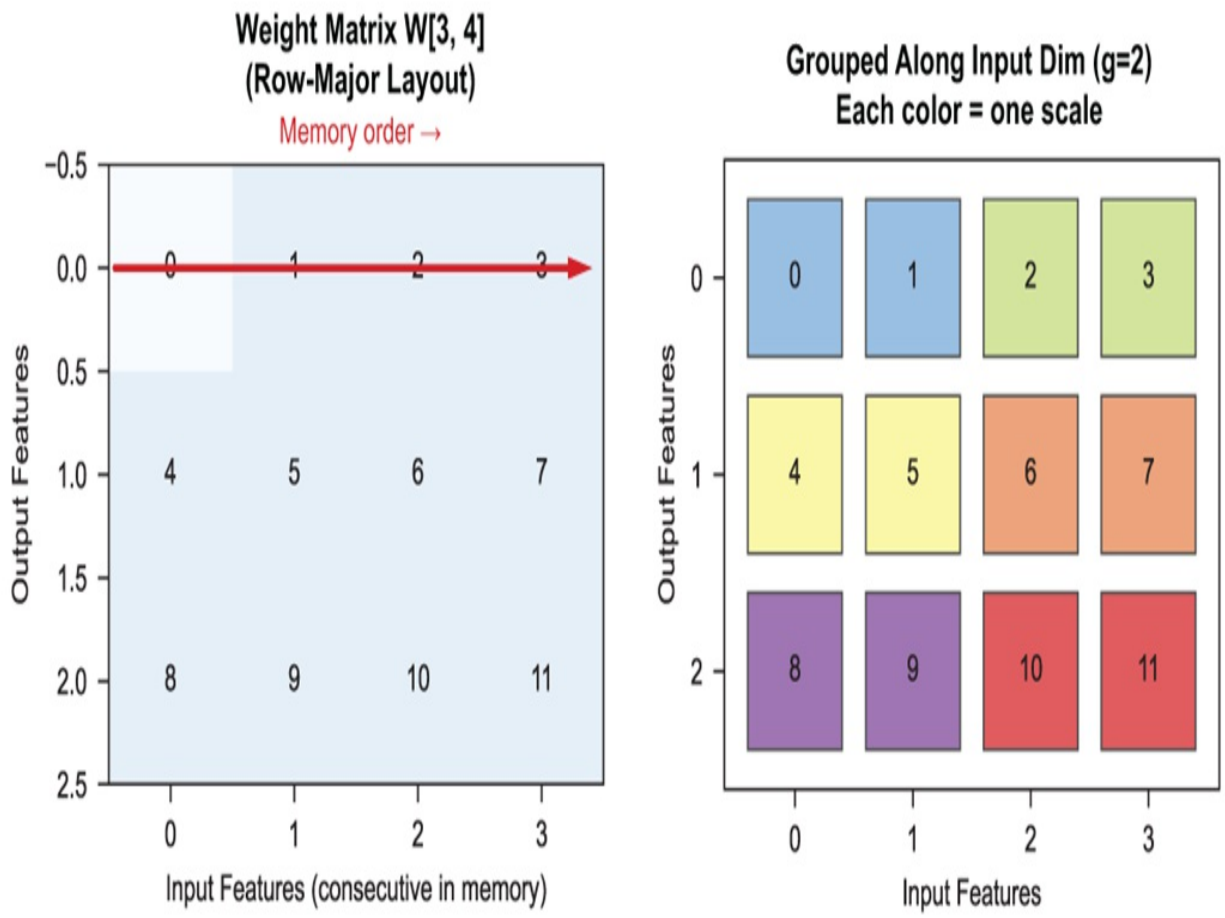
Memory layout: why it matters for kernels.

The choice of group size has profound implications for memory access patterns. GPUs achieve peak bandwidth with coalesced access—threads in a warp reading consecutive addresses in a single transaction. Scattered patterns collapse performance.

PyTorch tensors are row-major: elements along in_features are consecutive. Grouping along this axis aligns quantization with memory layout—load G quantized weights consecutively, load one scale, perform G multiply-accumulates, then scale the partial sum. Grouping along the output dimension would create strided access patterns, killing performance.

Figure 3.8 shows what the two layouts look like in memory: input-axis grouping keeps each group's elements contiguous (one cache line per group), while output-axis grouping spreads them across non-contiguous addresses (one cache miss per element).

Figure 3.8 Row-major layout means input features are consecutive in memory (left, arrow shows memory order). Grouping along this dimension (right, each color = one scale factor) enables coalesced loads: read G weights, load one scale, accumulate. Grouping along output features would scatter access across memory.



Consider how dequantization interleaves with the dot product. The scale s_g can be hoisted outside the inner sum over each group. This means we can load G quantized weights consecutively, load one scale, perform G integer multiply-accumulates, then scale the partial sum. G quantized values share one scale load—larger groups mean fewer scale loads but coarser quantization.

Block quantization: 2D grouping

NOTE

In much of the LLM quantization literature, 'block' and 'group' are used interchangeably to mean 1D partitioning (e.g., GPTQ, GGUF k-quants, bitsandbytes). Here we use 'block quantization' specifically to refer to a 2D extension.

Group quantization operates along a single axis. Block quantization extends this to 2D tiles. Listing 3.27 implements it by reshaping the weight matrix into a grid of block_rows × block_cols tiles, taking the absolute maximum within each tile as that tile's scale, then quantizing every element against the scale of the tile it belongs to.

Listing 3.27 Block quantization

```
def block_quantize(tensor, block_size=(32, 32), bits=4):
    rows, cols = tensor.shape
    block_rows, block_cols = block_size
    q_max = (1 << (bits - 1)) - 1
    tensor_blocked = tensor.reshape(
        rows // block_rows, block_rows, cols // block_cols, block
    ).permute(0, 2, 1, 3)
    abs_max = tensor_blocked.abs().amax(dim=(2, 3))
    scales = torch.clamp(abs_max / q_max, min=1e-8)
    quantized = torch.round(tensor_blocked / scales[:, :, None, N
    return quantized.clamp(-q_max - 1, q_max), scales
```

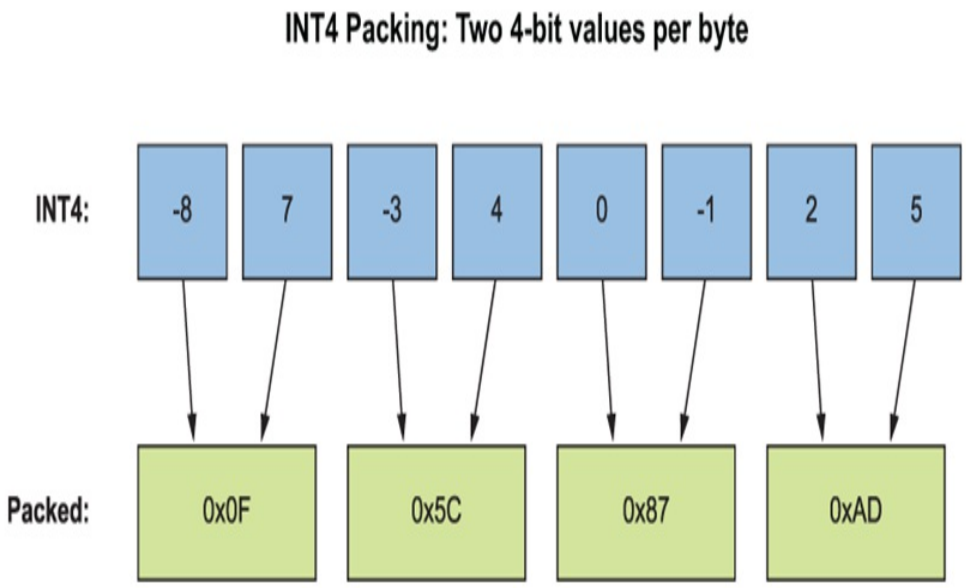
2D block quantization is uncommon in LLM inference, where 1D grouping dominates. Its primary application is FP4 training: NVIDIA's NVFP4 format uses 16×16 blocks so that scale factors remain valid under matrix transposition—essential for backpropagation. Activations and gradients still use 1D scaling.

Sub-byte packing

INT4 promises 2× the memory savings of INT8, but CPUs and GPUs operate on bytes, not nibbles (4 bits make a nibble). Two INT4 values must be packed into each byte.

Figure 3.9 illustrates the packing scheme: two INT4 values share each byte, with one in the high nibble and one in the low.

Figure 3.9 INT4 packing stores two values per byte. The first value occupies the high nibble (bits 4–7), the second the low nibble (bits 0–3). Unpacking requires bit shifts and masks before arithmetic—a cost that must be offset by memory bandwidth savings.



Listing 3.28 has three pieces. pack_int4 shifts each signed value from [-8, 7] to unsigned [0, 15], pairs adjacent values, and combines each pair into a single byte (the first in the high nibble, the second in the low). unpack_int4 reverses this with shifts and masks, then shifts back to signed. compute_memory_layout totals the bytes for packed INT4 weights and FP16 scales against the FP16 baseline for a typical 4096×4096 Q projection.

Listing 3.28 INT4 packing and memory layout

```
def pack_int4(values):
    """Pack pairs of INT4 values [-8,7] into bytes."""
    values_unsigned = (values.clamp(-8, 7) + 8).to(torch.uint8)
    paired = values_unsigned.reshape(*values.shape[:-1], -1, 2)
    return (paired[..., 0] << 4) | paired[..., 1]

def unpack_int4(packed):
    """Unpack bytes into INT4 value pairs."""
    high = (packed >> 4) & 0x0F
    low = packed & 0x0F
    unpacked = torch.stack([high, low], dim=-1).reshape(*packed.s
    return unpacked.to(torch.int8) - 8

def compute_memory_layout(out_features, in_features, group_size,
    num_groups = in_features // group_size
    quantized_bytes = (out_features * in_features * bits) // 8
    scale_bytes = out_features * num_groups * 2
    fp16_bytes = out_features * in_features * 2
```

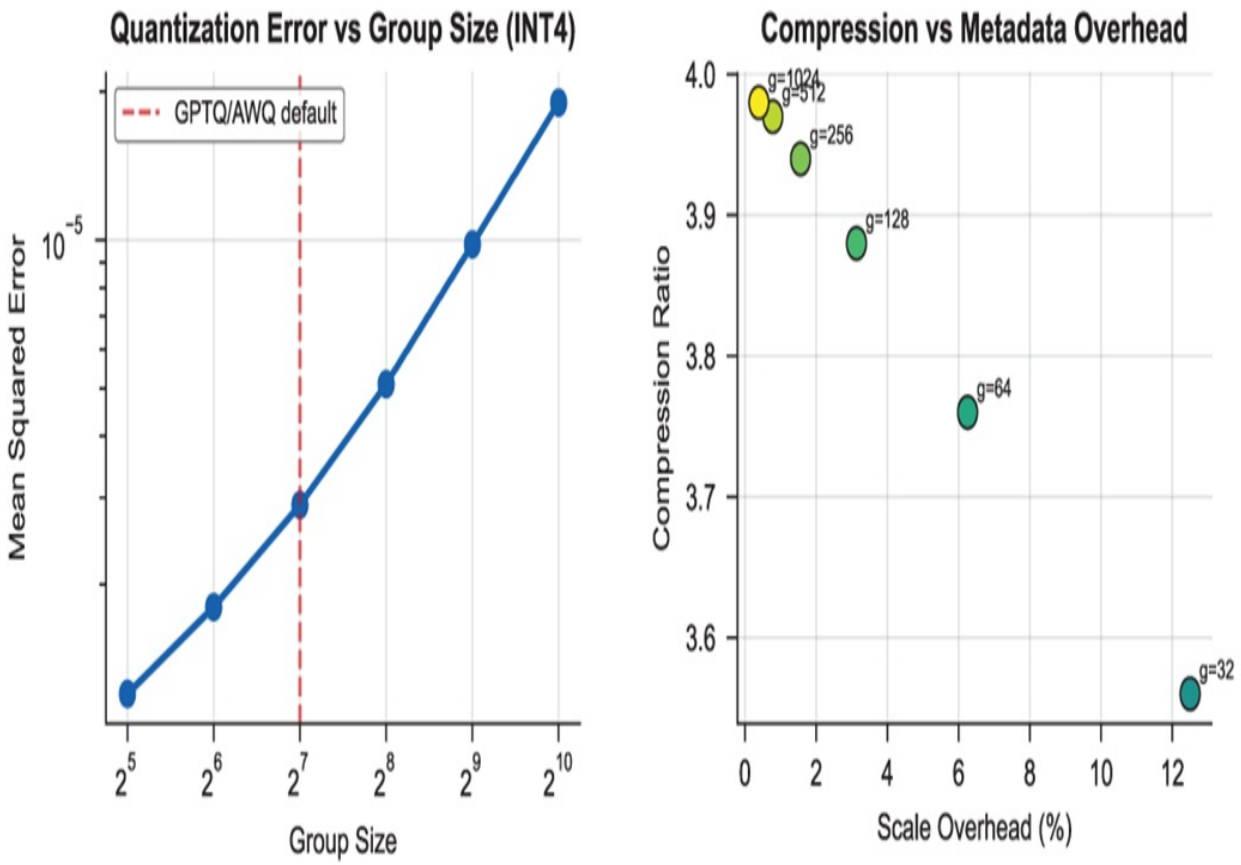
```
    total = quantized_bytes + scale_bytes
    return {
        'compression': fp16_bytes / total,
        'overhead_pct': scale_bytes / total * 100
    }
```

Output:

```
Memory Layout: q_proj layer [4096, 4096] (INT4, group_size=128)
--------------------------------------------------
Quantized weights:     8,388,608 bytes  (8 MB)
Scale factors:           262,144 bytes  (256 KB)
Total:                 8,650,752 bytes  (8.25 MB)
FP16 baseline:        33,554,432 bytes  (32 MB)
Compression ratio:          3.88x
Scale overhead:              3.0%
```

Packing interacts with group boundaries—efficient kernels require groups to align with byte boundaries. Group sizes that are multiples of 2 (for INT4) ensure clean alignment. The 3% scale overhead is a reasonable price for group-128's accuracy gains. Figure 3.10 quantifies the accuracy-overhead tradeoff across group sizes, revealing why 128 has become the industry default.

Figure 3.10 Smaller groups reduce MSE (left) but add scale overhead that erodes compression (right). Group-128 sits at the sweet spot: ~3% overhead for 2× lower error than group-256. Below g=64, overhead climbs steeply with diminishing accuracy returns.



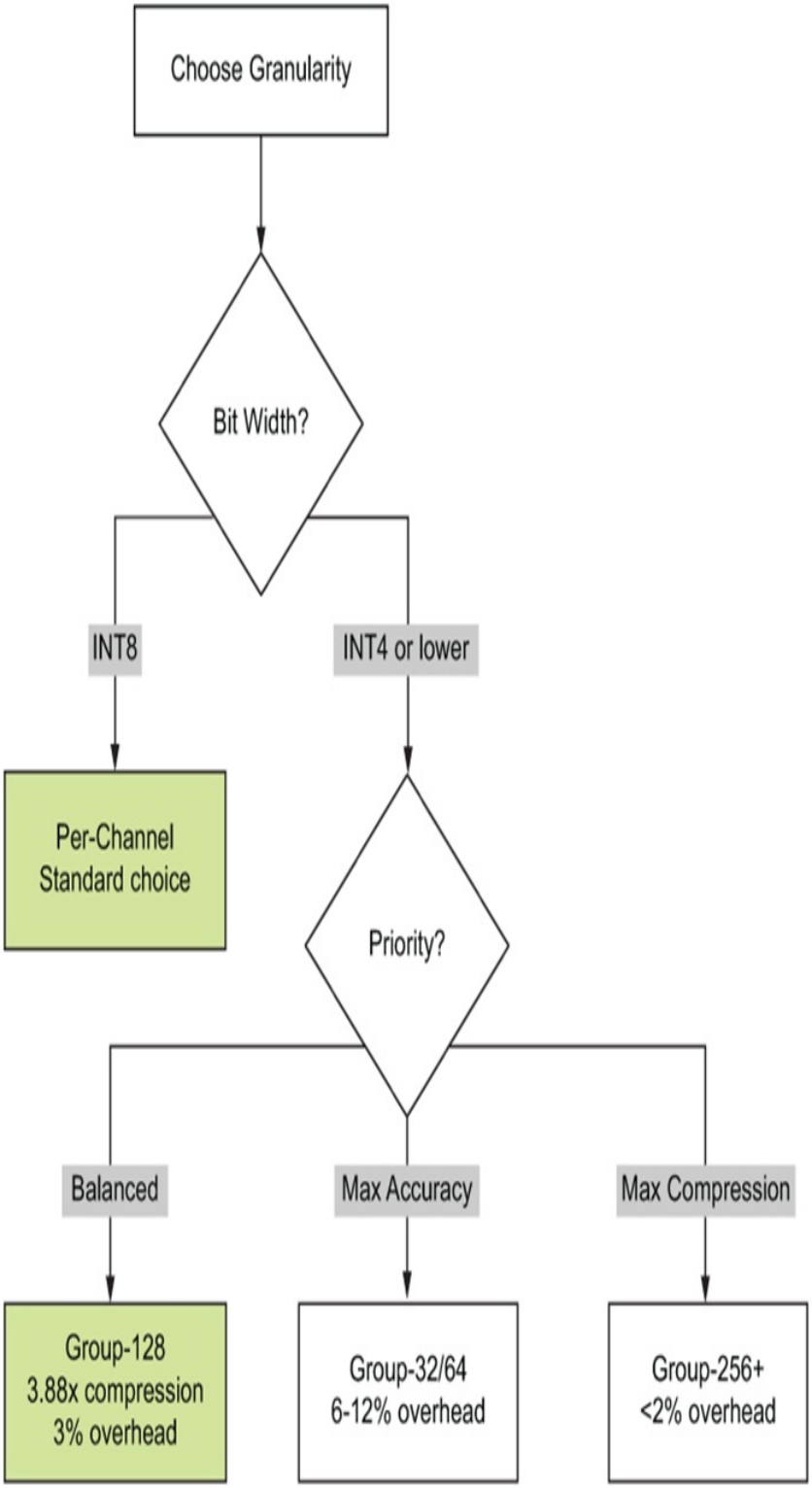
The points cluster into natural regions: group sizes ≥512 achieve near-4× compression with minimal overhead, while ≤64 trades compression for maximum accuracy.

Production guidelines

For INT4 weights, use group_size=128 as the default. Use 32 or 64 for accuracy-critical applications (6–12% overhead), or 256+ when every byte counts. For KV cache, use the asymmetric granularity from Section 3.4 (per-channel keys, per-token values)—group quantization is rarely needed since natural structure already provides good groupings.

For hardware compatibility, verify group size is a power of 2 and ≥32. Some accelerators have fixed block sizes—check documentation. Figure 3.11 distills these tradeoffs into a decision flowchart—start with bit width, then let your priority guide granularity.

Figure 3.11 Granularity decision tree. INT8 is simple: per-channel is the standard. INT4 requires choosing your priority—group-128 balances accuracy and compression, smaller groups maximize accuracy at higher overhead, larger groups maximize compression at some accuracy cost.



NOTE

The mental anchor for Group and Block quantization. Group quantization divides tensors into G-element groups with a separate scale each. Group-128 reduces INT4 MSE by ~2× over per-channel while adding only ~3% overhead—the industry default.

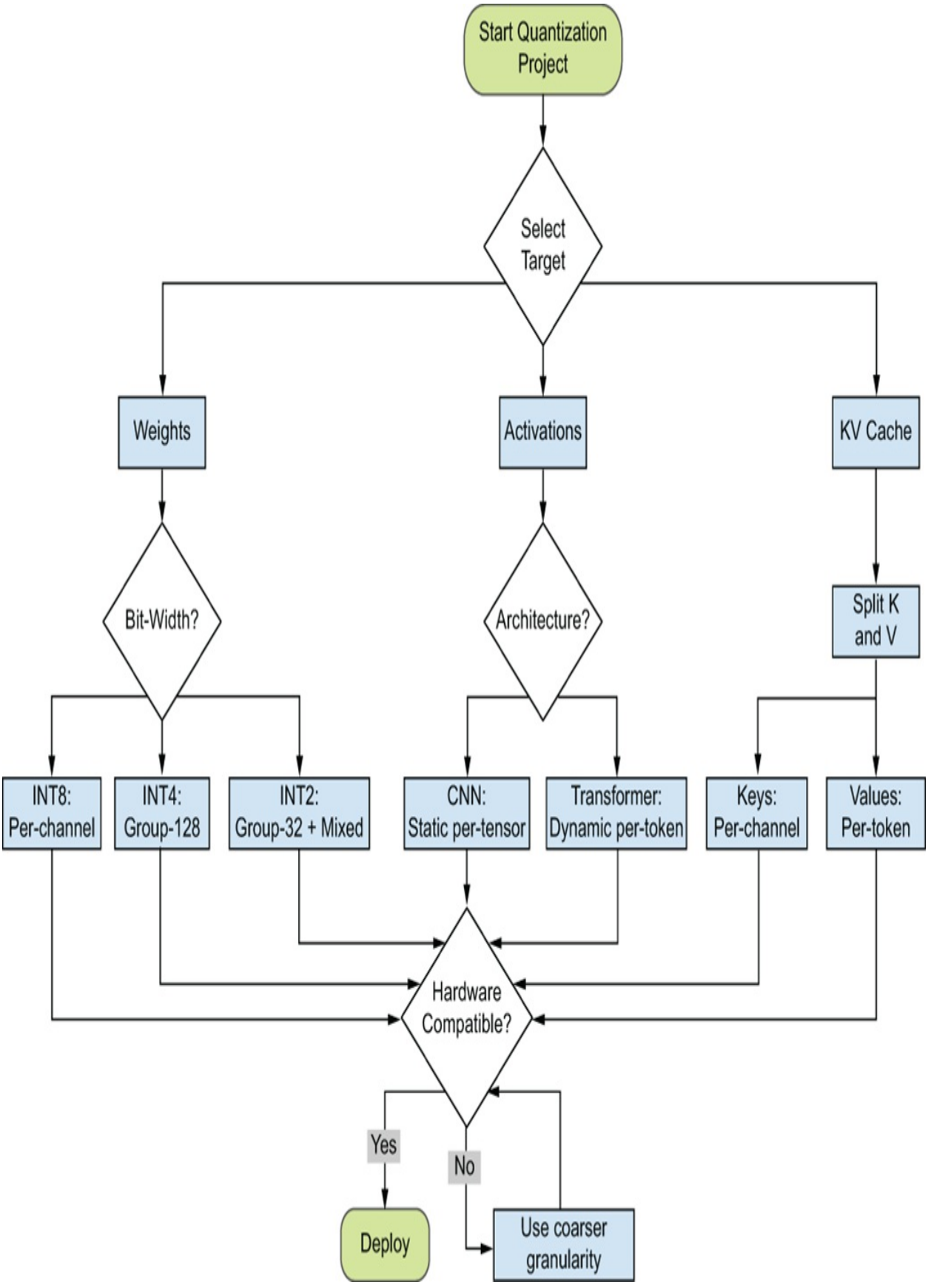
Memory layout and granularity must be co-designed. Grouping along the contiguous axis (input dimension for row-major linear layers) enables coalesced loads; misalignment kills performance. Sub-byte packing requires alignment between group and byte boundaries.

Reproducing these results: ch3/group_quantization_analysis.py. Representative blocks appear in the chapter; the full runnable version lives in the repo.

3.6 Follow a decision checklist for granularity

Now we consolidate everything into an actionable decision process—the checklist you'll consult when standing up a quantization pipeline. Three sequential questions drive the process: what to quantize? (target selection), at what granularity? (scale factor scope), and at what bit-width? (precision level). These interact—granularity depends on both target and bit-width. Figure 3.12 integrates everything into a master decision flowchart—target selection, granularity, bit-width, and the hardware compatibility gate that finalizes your choices.

Figure 3.12 End-to-end quantization decision tree, read top-to-bottom. Phase 1 (top): select which tensors to quantize—weights are always included, activations add compute speedup, KV cache is context-length dependent. Phase 2 (middle): for each target, select granularity using bit-width as the primary fork—INT8 paths favor per-channel/per-token, INT4 paths favor group-128. Phase 3 (bottom): architecture-specific overrides (e.g., first/last layers in FP16). All paths converge on the hardware compatibility gate (right)—if your runtime doesn't support the chosen scheme, fall back to the next coarser granularity.


Start Quantization Project
Select Target
Weights
Activations
KV Cache
Bit-Width?
Architecture?
Split K and V
INT8: Per-channel
INT4: Group-128
INT2: Group-32 + Mixed
CNN: Static per-tensor
Transformer: Dynamic per-token
Keys: Per-channel
Values: Per-token
Hardware Compatible?
Yes
No
Deploy
Use coarser granularity

Four factors recur at every decision point. *Model sensitivity*: attention-heavy models (transformers) are more sensitive than spatially redundant models (CNNs), requiring finer granularity. *Deployment latency budget*: real-time inference (<10ms) may demand aggressive INT4, while batch processing can afford the safety of INT8. *Hardware support*: not all runtimes support per-channel activations or group quantization—verify before committing (Phase 4). *Accuracy tolerance*: classification tasks with clear decision boundaries tolerate more quantization error than generation tasks where small logit shifts compound across tokens. Keep these factors in mind as you work through each phase below.

Phase 1: Target selection

- Weights: always quantize linear and conv layers. Exceptions: embedding tables and classification heads in FP16, bias terms in FP32 (negligible memory).
- Activations: quantize for compute speedup, not just memory savings. Requires calibration data; check for outlier channels in transformers.
- KV cache (transformers only): optional under 2K context, recommended over 8K, essential over 32K.

Phase 2: Granularity selection

Table 3.8 summarizes the recommended granularity for each target and bit-width combination:

Table 3.8 Recommended granularity by target and bit-width

Target	INT8	INT4	INT2
Weights (Linear/Conv)	Per-channel (output axis)	Group-128 (input axis)	Group-32 (+ outlier channels in FP16)
Weights (Embeddings)	Per-row or Per-channel	Group-128	Group-32 + Mixed precision
Activations	Per-tensor (CNN) or Per-token	Per-token (dynamic)	Not recommended

	(Transformer)		
KV Cache Keys	Per-channel	Per-channel	Research only
KV Cache Values	Per-token	Per-token	Research only

Phase 3: Architecture-specific considerations

For CNNs (ResNet, EfficientNet, YOLO), per-channel INT8 weights are usually sufficient, and static calibration works well for typical deployments. The first conv layer may need FP16 since it interfaces directly with raw pixel values. Depthwise convolutions use per-channel along the output axis, same as standard conv.

For transformers (BERT, GPT, LLaMA, Mistral): per-channel INT8 or group-128 INT4 for weights. Per-token dynamic quantization for activations—outlier channels are severe. Attention projections (Q, K, V, O) are more sensitive than FFN; keep LayerNorm in FP32 (tiny footprint, high sensitivity). Use asymmetric KV cache granularity (per-channel keys, per-token values).

For vision transformers (ViT, DeiT, Swin), treat the patch embedding layer like a first conv layer (may need FP16). Attention layers follow the same guidance as language transformers. The [CLS] token can accumulate outsized activations—watch for clipping.

Phase 4: Hardware verification

Runtimes vary in which schemes they accept, and an unsupported scheme silently degrades to a coarser one or to FP32 at load time. Before finalizing the configuration from Phases 1 through 3, verify support for your chosen granularity against the target runtime. Table 3.9 summarizes the matrix across the five runtimes covered later in the book.

Table 3.9 Runtime support matrix

Runtime	Per-channel weights	Group Quantization	Per-token Activations
TensorRT	✓ Supported	✓ Supported	✓ Supported
ONNX Runtime	✓ Supported	✓ Supported	✓ Supported
llama.cpp	✓ Via GGUF	✓ Native (Q4_K, Q5_K)	N/A (weights-only)
TFLite	✓ Supported	Limited	Per-tensor only
PyTorch (eager)	✓ Supported	✓ Supported	✓ Supported

Common pitfalls and quick fixes

- Accuracy collapses uniformly: move to per-channel (weights) or per-token (activations). No speedup: profile operators to verify INT8/INT4 kernels—silent FP32 upconversion is common.
- Attention corrupted: switch KV cache to asymmetric granularity. First layer outputs garbage: keep in FP16 or widen calibration range.
- Single layer kills accuracy: mixed-precision for top 0.1% magnitude channels. Group quantization slow: power-of-2 group sizes ≥32, aligned with hardware tiles.

3.7 Summary

- Weights are the safest target—static, zero-centered, tolerant of perturbation. Even INT4 often preserves quality with appropriate granularity.
- Activations unlock compute speedup but require calibration. Static works for CNNs; dynamic per-token is essential for transformers.
- The KV cache dominates long-context LLM memory, exceeding model weights at 128K context. Keys need per-channel quantization; values need per-token.
- Per-channel is the INT8 standard (3–9× better MSE than per-tensor). Group-128 is the INT4 standard (~2× better than per-channel, ~3%

overhead).
- Memory layout must align with granularity—group along the contiguous axis for coalesced loads.
- The decision framework: select targets, choose granularity by bit-width, apply architecture-specific adjustments, verify hardware compatibility.

4 Applying Post-Training Quantization and Calibration

This chapter covers

- Deciding when post-training quantization is sufficient for your deployment
- Building production-faithful calibration sets
- Range estimation algorithms and their trade-offs
- Applying validation protocols

Chapter 3 established what to quantize—weights, activations, KV cache—and at what granularity—per-tensor, per-channel, or group-wise. You left with a decision checklist that tells you which tensors deserve attention and how finely to slice the scale factors.

But knowing *what* to quantize is not the same as knowing *what data to derive your mapping from*. The scale factor S and zero-point Z that define your quantization grid are determined by the activations the model encounters — and those activations depend on the data, not the model. Derive them from calibration data that doesn't match production, and your carefully chosen granularity scheme produces garbage in deployment. Derive them from data that does, and a model quantized in minutes can match original accuracy.

This is the domain of calibration: the art of observing your model on representative data to determine the optimal quantization parameters. And the good news is that for most practitioners, calibration is enough. You don't need to retrain. You don't need gradients. You need good data, the right range estimation algorithm, and a validation protocol that catches problems before they hit production.

Here we show you how to do post-training quantization (PTQ) right. We start by establishing when PTQ — post-training quantization, where you derive quantization parameters from calibration data without touching the model's

weights — is sufficient, saving you weeks of QAT (quantization-aware training, where you retrain the model with quantization simulated in the forward pass; covered in Chapter 5) when it's unnecessary. We then build production-faithful calibration sets, explore range estimation algorithms (absolute max, percentile, MSE, KLD), apply correction techniques for stubborn accuracy gaps, and define a repeatable validation protocol. We'll then close out the lesson with a catalog of common pitfalls and their fast fixes to help cement the knowledge.

4.1 Know when post-training quantization is enough

Post-training quantization—where you take a trained model, observe its behavior on calibration data, and derive quantization parameters without any gradient-based optimization—is the practitioner's first choice for good reason. It's fast (minutes to hours, not days), it's cheap (no GPU cluster for training), and it is sufficient surprisingly often.

The alternative, quantization-aware training (QAT), retrains the model with fake quantization operators inserted into the forward pass. QAT can recover accuracy that PTQ cannot—but it costs training compute, requires hyperparameter tuning, and adds engineering complexity. Reaching for QAT when PTQ would suffice is like calling in a surgeon for a paper cut.

PTQ's success depends on four factors that interact in predictable ways. Understanding these lets you predict—before you even try—whether PTQ will work for your model.

These four factors are:

- *Model architecture*—how inherently robust the model is to small perturbations.
- *Target bit-width*—the quantization precision you're aiming for.
- *Model size and redundancy*—whether the model has enough spare capacity to absorb quantization error.
- *Accuracy tolerance*—how much degradation your application can accept.

We'll examine each factor in turn, then combine them into a practical decision framework (Figure 4.2) that you can apply mechanically to any new model. These factors compound rather than add: a small attention-heavy model (architecture and size both unfavorable) targeting INT4 (aggressive bit-width) in a medical imaging system (low accuracy tolerance) almost certainly needs QAT — no calibration trick recovers all four headwinds at once. Conversely, a large CNN targeting INT8 with a 2% tolerance is the "PTQ-trivial" corner of the space.

Model architecture

Some architectures are inherently more quantization-friendly than others. Models trained with dropout, weight decay, and data augmentation develop robust representations that tolerate the small perturbations quantization introduces—ResNets, EfficientNets, and most production CNN architectures fall into this category.

Models with smooth, bounded activations quantize more easily than those with extreme values, but attention mechanisms in transformers produce activation outliers 10-30× larger than the median (as quantified in Section 3.3). Residual connections propagate quantization errors through the entire network; more skip connections (as in transformers) create more pathways for error accumulation.

To quantify these sensitivity differences, listing 4.1 runs a controlled experiment. We take three models from different architecture families (ResNet-18, MobileNetV2, BERT-base), inject Gaussian noise whose standard deviation is 1% of each weight tensor's mean absolute value, and measure how much the model's output shifts. The injected noise stands in for the per-weight perturbation that uniform quantization introduces, and the resulting relative output change is our proxy for architectural sensitivity. Ten trials average out the random component so the architecture effect dominates the signal.

Listing 4.1 Measuring architecture noise sensitivity

```
def measure_noise_sensitivity(model, input_tensor, noise_scale=0.
    model.eval()
```

```
    with torch.no_grad():
        baseline = model(input_tensor)
        if hasattr(baseline, 'last_hidden_state'):
            baseline = baseline.last_hidden_state

        output_changes = []
        for _ in range(num_trials):
            original_weights = {}
            for name, param in model.named_parameters():
                original_weights[name] = param.data.clone()
                noise = torch.randn_like(param) * noise_scale * p
                param.data.add_(noise)

            noisy_output = model(input_tensor)
            if hasattr(noisy_output, 'last_hidden_state'):
                noisy_output = noisy_output.last_hidden_state

            relative_change = (noisy_output - baseline).abs().mea
            output_changes.append(relative_change.item())

            for name, param in model.named_parameters():
                param.data.copy_(original_weights[name])

        return sum(output_changes) / len(output_changes)

torch.manual_seed(42)
resnet = resnet18(pretrained=False)
mobilenet = mobilenet_v2(pretrained=False)
bert = BertModel.from_pretrained('bert-base-uncased')
print(f"ResNet-18:   {measure_noise_sensitivity(resnet, torch.ran
print(f"MobileNetV2: {measure_noise_sensitivity(mobilenet, torch.
print(f"BERT-base:   {measure_noise_sensitivity(bert, torch.randi
```

Output:

```
ResNet-18:   0.0257
MobileNetV2: 0.0023
BERT-base:   0.0742
```

The results reveal an important nuance: MobileNetV2 shows the lowest sensitivity (~0.2%) despite being an "efficient" architecture—its depthwise separable convolutions create inherent regularization. ResNet-18 is moderate (~2.6%), while BERT shows dramatically higher sensitivity (~7.4%). This 30× gap between MobileNetV2 and BERT translates directly to PTQ

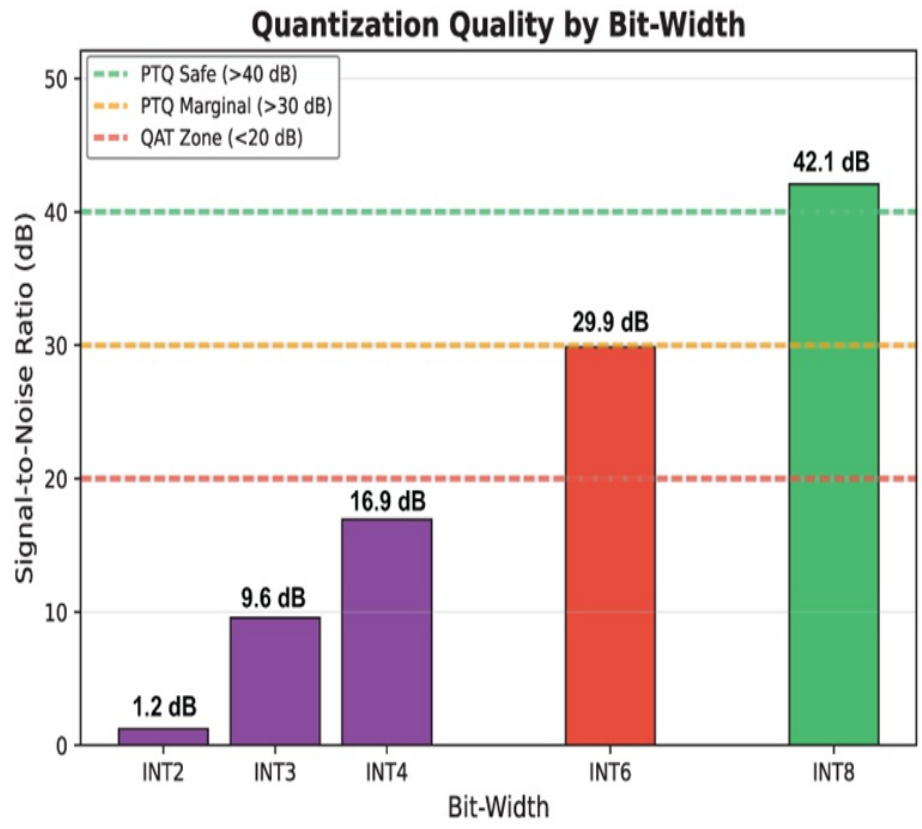
difficulty—not because of architecture type, but because of how attention mechanisms amplify small perturbations through softmax and residual streams.

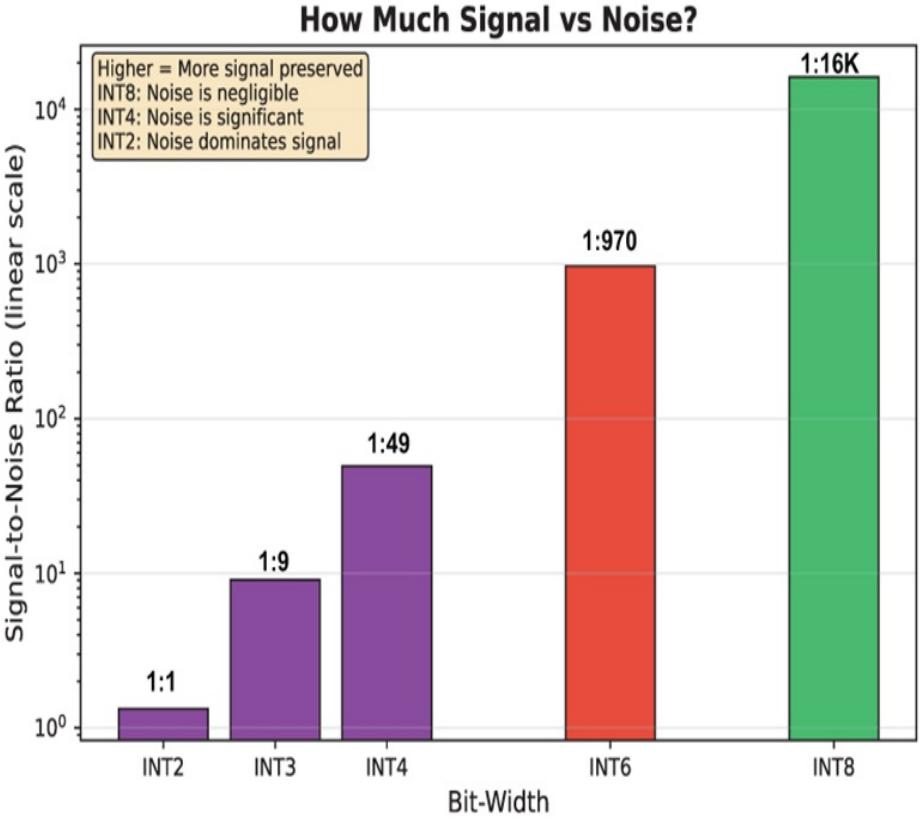
Architecture tells you how sensitive a model is to quantization noise—but the amount of noise itself depends on how aggressively you quantize.

Target bit-width

The relationship between bit-width and PTQ success is not linear—it's a cliff. Figure 4.1 shows why: signal-to-noise ratio doesn't degrade gracefully as you reduce precision—it plummets.

Figure 4.1 Left: Signal-to-Noise ratio by bit-width with PTQ viability zones. Right: Signal-to-noise ratio as intuitive ratios


Quantization Quality by Bit-Width
PTQ Safe (>40 dB)
PTQ Marginal (>30 dB)
QAT Zone (<20 dB)
Signal-to-Noise Ratio (dB)
50
40
30
20
10
0
1.2 dB
9.6 dB
16.9 dB
29.9 dB
42.1 dB
INT2
INT3
INT4
INT6
INT8
Bit-Width


How Much Signal vs Noise?
Higher = More signal preserved
INT8: Noise is negligible
INT4: Noise is significant
INT2: Noise dominates signal
Signal-to-Noise Ratio (linear scale)
10^4
10^3
10^2
10^1
10^0
1:1
1:9
1:49
1:970
1:16K
INT2
INT3
INT4
INT6
INT8
Bit-Width

At INT8, quantization noise is ~16,000× smaller than the signal—like background static during a concert. At INT4, noise is only ~50× smaller—more like trying to hold a conversation at that same concert. This dramatic drop in signal-to-noise ratio explains why INT8 PTQ succeeds almost universally: the quantization grid is fine enough that rounding errors rarely matter. INT4, by contrast, has so few levels that per-tensor quantization wastes most of them on outliers, leaving the bulk of the distribution poorly represented. This is precisely why INT4 requires group quantization to succeed with PTQ—smaller groups mean each set of weights gets its own scale factor, effectively recovering resolution that the coarse grid takes away.

Bit-width determines how much noise quantization introduces, and architecture determines how sensitive the model is to it — but the model's capacity to absorb that noise matters just as much.

Model size and redundancy

Larger models are generally more tolerant of quantization, not less. This seems counterintuitive—wouldn't more parameters mean more opportunities for error?

The explanation lies in overparameterization. Large models have more redundancy—the same information is encoded across many weights, so corrupting a few doesn't destroy the signal. Small models use every parameter more efficiently; each weight carries more load, and perturbing it causes larger output changes.

Interestingly, our noise sensitivity experiment showed MobileNetV2 as less sensitive than ResNet-18 despite having fewer parameters (3.4M vs 11.7M). This highlights that architecture design matters as much as size—MobileNetV2's depthwise separable convolutions create implicit regularization that improves quantization robustness. The key insight: model size is a useful heuristic, but architecture-specific properties can override it.

Accuracy tolerance

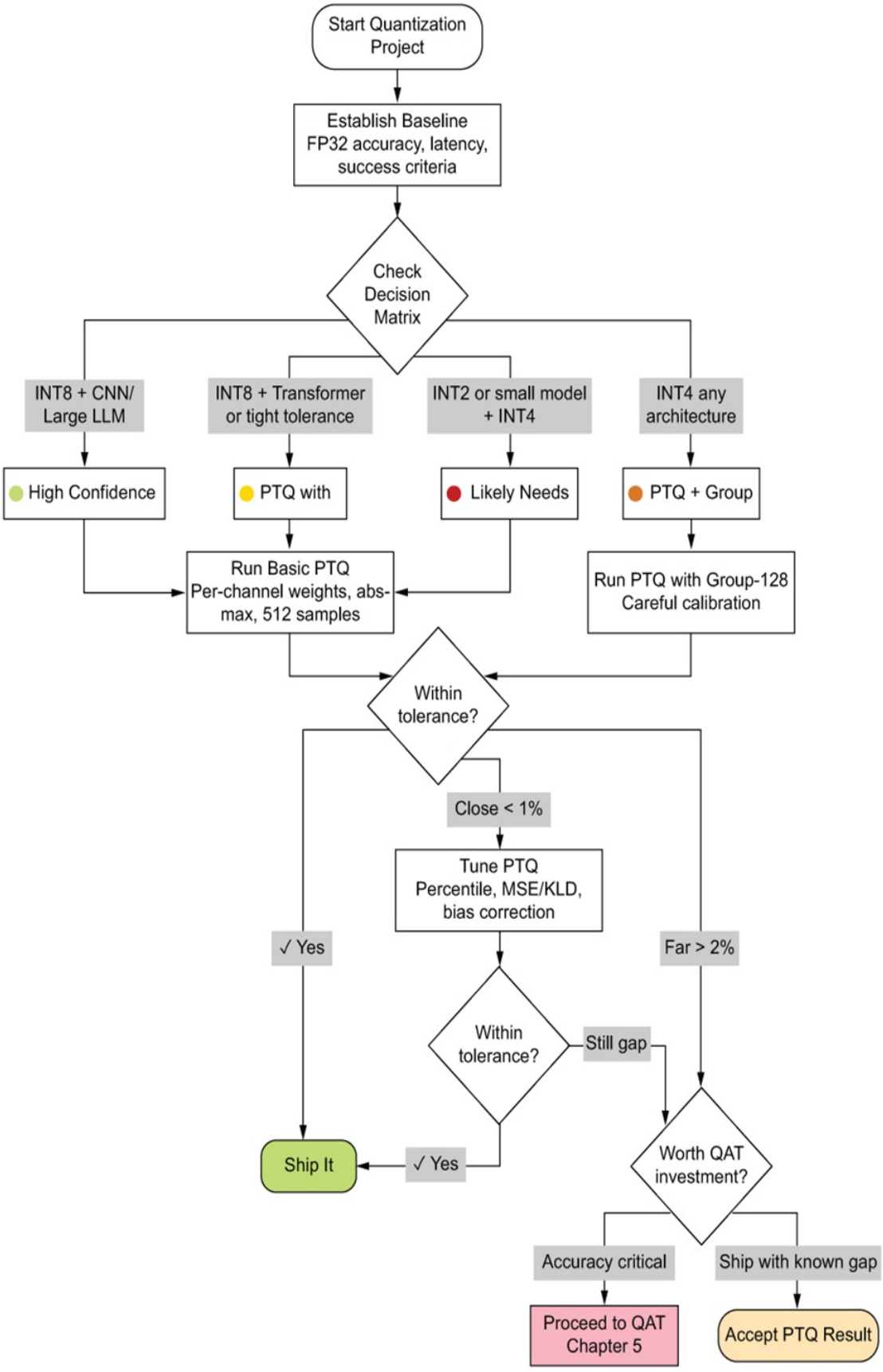
Perhaps the most important factor to note is how much loss in accuracy your use case can tolerate. This isn't a technical property — it's a business constraint. The same 0.8% PTQ gap is acceptable for a recommendation system with a 2% error budget but disqualifying for a medical imaging classifier with a 0.5% one — the latter requires QAT regardless of how clean the PTQ pipeline looks. A real-time speech recognition system might happily accept slightly higher word error rates for the latency win from INT8.

Since accuracy tolerance is often a business decision, not an engineering one, establish your budget and get stakeholder sign-off before you quantize; a "failed" quantization to an ML engineer may be perfectly acceptable to product, and a passing one may still be disqualifying for compliance.

The decision framework

The four factors above combine into a practical decision framework, shown in Figure 4.2. The flowchart starts with your target bit-width—the single strongest predictor of PTQ success. If you're targeting INT8, the path is short: basic PTQ with per-channel weight quantization handles the vast majority of cases. If accuracy falls short, you escalate through advanced calibration methods (percentile, MSE-optimal) before considering QAT. For INT4 and below, the flowchart branches based on model size and architecture—large models with redundancy tolerate aggressive quantization through group schemes, while small or attention-heavy models are routed toward QAT earlier.

Figure 4.2 PTQ decision framework flowchart. Starting from the target bit-width at the top, the reader follows branching decisions through model architecture checks, calibration method selection, and accuracy validation gates. Each terminal node recommends either shipping the PTQ result or escalating to QAT.


Start Quantization Project
Establish Baseline FP32 accuracy, latency, success criteria
Check Decision Matrix
INT8 + CNN/ Large LLM
INT8 + Transformer or tight tolerance
INT2 or small model + INT4
INT4 any architecture
High Confidence
PTQ with
Likely Needs
PTQ + Group
Run Basic PTQ Per-channel weights, abs-max, 512 samples
Run PTQ with Group-128 Careful calibration
Within tolerance?
Close < 1%
Tune PTQ Percentile, MSE/KLD, bias correction
✓ Yes
Far > 2%
Within tolerance?
Still gap
Ship It
✓ Yes
Worth QAT investment?
Accuracy critical
Ship with known gap
Proceed to QAT Chapter 5
Accept PTQ Result

The key insight from this framework: most practitioners stop after basic PTQ, and rightly so. Per-channel weights with absolute-max calibration solves the majority of INT8 problems. When that falls short, the framework guides you through a structured escalation—trying advanced calibration methods like percentile clipping or MSE-optimal range estimation (covered in Section 4.3) before moving to mixed-precision strategies. Only when these intermediate steps fail does QAT enter the picture. In practice, this means QAT is genuinely rare—reserved for sub-4-bit quantization or unusually sensitive models where even the best calibration cannot close the accuracy gap.

The mental anchor for PTQ:

PTQ works more often than people expect. For INT8 quantization of standard architectures, PTQ with proper calibration succeeds in the vast majority of cases. Don't default to QAT — start with PTQ. Bit-width is the primary determinant: INT8 is almost always PTQ-safe, INT4 requires group quantization, and below INT4, QAT becomes necessary.

4.2 Build a calibration set that actually represents production

Even the best PTQ algorithm fails if it calibrates on the wrong data. The scale factors and zero-points that define your quantization grid are only as good as the data that produced them.

The distribution mismatch problem

Calibration data has one job: reveal the activation ranges your model will encounter in production. If calibration data differs systematically from production data, the estimated ranges will be wrong—either too wide (wasting precision) or too narrow (clipping common values).

Consider a sentiment analysis model trained on product reviews, calibrated on polite customer service examples. In production, it encounters angry social media posts. Activation magnitudes spike beyond the calibrated range,

and the model misclassifies negative sentiment as neutral—not because quantization is lossy, but because the calibration data promised activations would stay small.

The same pattern appears in vision: a model calibrated on well-lit studio photos will miscalibrate for security camera footage. In LLMs: a model calibrated on short factual queries will clip activations when processing long technical documents. The failure mode is always the same—production data breaks promises that calibration data made.

What "representative" actually means

"Representative calibration data" sounds simple, but production data has structure that's easy to miss. Temporal patterns mean a recommendation system sees different queries at 9 AM versus 9 PM—calibrate only on daytime traffic, and nighttime accuracy suffers. User segments differ: enterprise and consumer users query differently, so a B2B chatbot calibrated on consumer questions miscalibrates for enterprise jargon. Edge cases that matter—the 1% of unusual inputs—may be the most important for fraud detection, safety-critical queries, or high-value customers, but random sampling under-weights these.

The goal isn't to capture every possible input—it's to capture the range of activation patterns. Think of calibration as exploring the activation space, not statistical sampling. Figure 4.3 breaks this into a repeatable workflow—from sourcing raw production data through validation—that ensures your calibration set covers the activation patterns that actually matter.

Figure 4.3 Workflow for preparing a calibration set. The process flows from top to bottom: source production data from logs or traffic samples, apply stratified sampling to cover user segments and edge cases, validate distribution coverage against known activation patterns, and iterate if coverage gaps are found. Each stage includes a quality gate that prevents poorly representative data from reaching calibration.

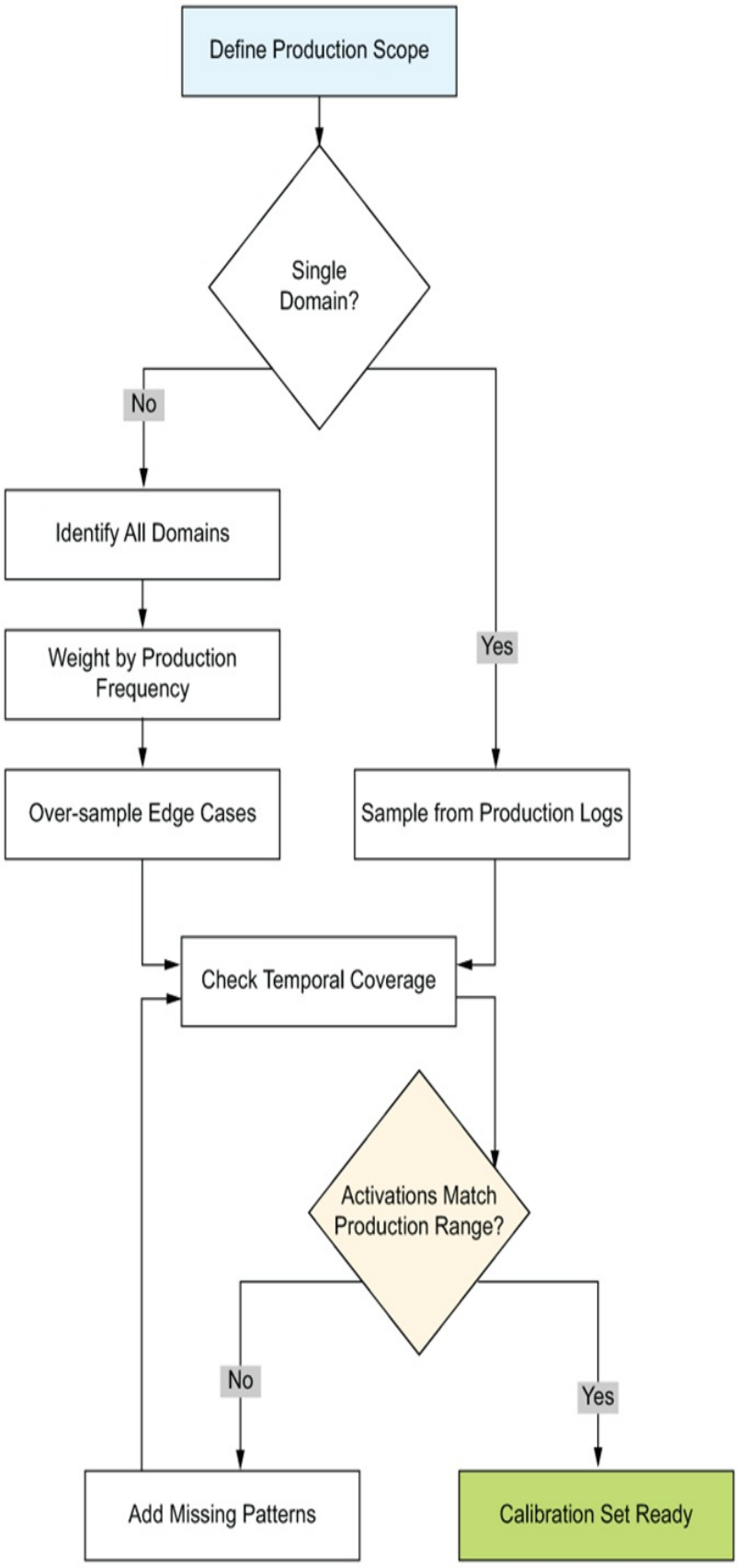
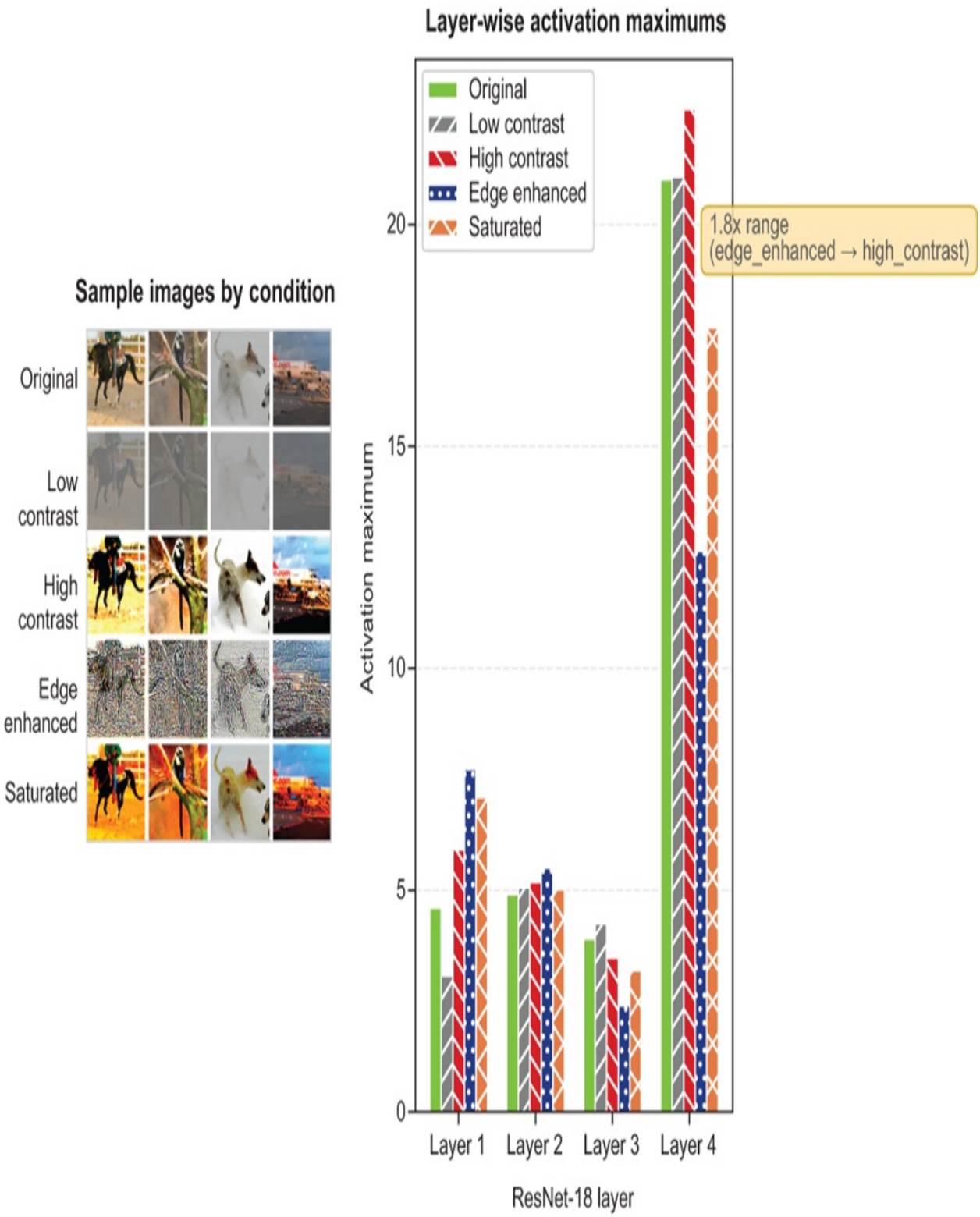

Define Production Scope
Single Domain?
No
Yes
Identify All Domains
Weight by Production Frequency
Over-sample Edge Cases
Sample from Production Logs
Check Temporal Coverage
Activations Match Production Range?
No
Yes
Add Missing Patterns
Calibration Set Ready

Image classification models need calibration data covering diverse visual conditions: lighting, contrast, object scales, and image quality. To demonstrate this, we apply controlled transformations to real images and measure how activation patterns change.

The companion script ch4/vision_calibration_analysis.py loads images from STL-10 (96×96, clearer than CIFAR's 32×32) dataset, applies deliberate transformations, and measures ResNet-18 activations. The transformations are intentionally exaggerated to make activation differences visible—real production variation is subtler but follows the same patterns.

Figure 4.4 shows the results of this analysis. The left panel displays sample images under four conditions—original, low-light, high-contrast, and noisy—illustrating the visual transformations applied. The right panel plots the corresponding layer-wise activation maximums through ResNet-18's layers, revealing how different input conditions shift the activation ranges that calibration must capture.

Figure 4.4 Left: Sample images under four conditions (original, low-light, high-contrast, noisy) showing the visual transformations applied to STL-10 images. Right: Layer-wise activation maximums through ResNet-18, where each line corresponds to one condition. Notice how low-light images produce consistently lower activation ranges while high-contrast images push activation maximums higher in deeper layers—calibrating on only one condition would systematically mis-estimate the ranges encountered in production.



Output from the analysis:

```
Activation Ranges by Image Condition (ResNet-18):
```

Condition	Layer1	Layer2	Layer3	Laye
original	4.61	4.91	3.90	21.
low_contrast	3.07	5.06	4.25	21.
high_contrast	5.91	5.18	3.47	22.
edge_enhanced	7.73	5.50	2.40	12.
saturated	7.09	5.02	3.18	17.
Max/Min Ratio	2.5x	1.1x	1.8x	1.

The results reveal non-obvious patterns. High contrast produces the largest Layer 4 activations (22.60)—the final layer sees more extreme features when input contrast is boosted. Edge enhanced has the highest Layer 1 activations (7.73) but the lowest Layer 4 activations (12.64)—early convolutional layers respond strongly to high-frequency edge content, but the artificial noise doesn't produce meaningful high-level features. Low contrast barely affects deeper layers—the network compensates through normalization, producing similar Layer 4 activations (21.07) to original images despite washed-out inputs.

The 1.8× range in Layer 4 means calibrating only on "normal" images would set scale factors that clip activations from high-contrast inputs. In production, this translates to degraded accuracy on challenging images—exactly when you need the model to perform well.

NOTE

Match calibration data to deployment conditions. If your model runs in a constrained environment (factory line, constant lighting), don't artificially diversify; calibrate on the conditions it will actually see. If it deploys to diverse environments (consumer phones, security cameras), include deliberately challenging examples spanning lighting extremes, motion blur, compression artifacts, and edge cases. In both cases a few hundred well-chosen samples beat thousands of redundant ones.

Building calibration sets for LLMs

Language models present a different challenge. The input space is vast, and

different linguistic patterns produce different activation magnitudes. For LLMs, representative calibration means covering prompt length variation (short and long prompts activate differently; KV cache behavior changes with sequence length), linguistic diversity (formal text, casual text, code, math, and mixed languages produce distinct patterns), task diversity (summarization, QA, creative writing, and reasoning stress different model components), and structured content (code, mathematical notation, and formatted text often produce larger activations than prose).

Listing 4.2 puts this to the test—we define prompt categories that span these dimensions, then measure how each category's activation magnitudes differ across layers of TinyLlama-1.1B.

Listing 4.2 LLM calibration prompt categories

```
calibration_prompts = {
    'short_factual': ["What is the capital of France?", ...],
    'long_context': ["The following is a detailed analysis...", .
    'code': ["def quicksort(arr):\n    if len(arr) <= 1:", ...],
    'reasoning': ["If all roses are flowers, and some flowers..."
    'creative': ["Write a haiku about artificial intelligence:",
    'multilingual': ["Translate to French: The weather is beautif
    'adversarial': ["!!!URGENT!!!", "aaaaaaaaaa", "🔥🚀💯🎉✨",
}

with torch.no_grad():
    outputs = model(**inputs, output_hidden_states=True)
for layer_idx, hidden_state in enumerate(outputs.hidden_states):
    abs_max = hidden_state.abs().max().item()
```

The companion script ch4/llm_calibration_builder.py loads TinyLlama-1.1B, tokenizes 20 prompts from each of the seven categories, runs forward passes with `output_hidden_states=True`, and records the per-layer absolute maximum activation. Layers 6 and 11 (the deepest of TinyLlama's 22) show the divergence between categories most clearly. The output below aggregates 140 prompts in total, sorted by L11 maximum activation:

```
Activation Analysis by Prompt Category:
============================================================
Category             | Samples | Avg Len |     L6 Max |    L11 Ma
------------------------------------------------------------
```

```
code                |      20 |      48 |      140.6 |      186.
long_context        |      20 |     112 |      140.6 |      171.
adversarial         |      20 |      19 |      140.6 |      168.
reasoning           |      20 |      45 |      140.6 |      139.
short_factual       |      20 |       9 |      140.6 |      139.
creative            |      20 |      12 |      140.6 |      139.
multilingual        |      20 |      14 |      140.6 |      139.
------------------------------------------------------------------
Combined coverage   |     140 |      -- |      140.6 |      186.
==================================================================
Key Insight: 'code' produces the largest activations (186.4).
Range across categories: 1.34x (multilingual → code).
```

Interpreting with appropriate caveats

This experiment uses TinyLlama (1.1B parameters) with hand-crafted prompts—a controlled demonstration, not production-grade calibration. Several limitations apply: to reach 20 samples per category, some prompts are duplicated, inflating sample counts without adding activation diversity; 20 prompts per category barely scratches the surface of real-world input diversity; and TinyLlama's activation patterns may differ significantly from larger models.

Despite these limitations, the directional finding is consistent with what production calibration sets show: code produces the largest activations (186.4 in L11) — structured syntax with special characters, indentation, and keywords creates distinct activation signatures. Long context and adversarial inputs also push higher than conversational text.

The production principle holds: A calibration set consisting only of short conversational queries would miss the upper end of the activation range. Real calibration sets from GPTQ (Frantar et al., 2022) and AWQ (Lin et al., 2023) authors use curated samples from diverse corpora — Wikipedia, GitHub, C4, multilingual datasets — with careful deduplication and quality filtering.

The calibration size trade-off

How many samples do you need? The answer from production systems: less than you might think, but with caveats.

GPTQ uses 128 samples by default. AWQ uses 128-256. The llama.cpp quantization scripts default to 128-512. These aren't arbitrary—scale factor estimation stabilizes quickly once you've captured the activation range extremes.

To see how quickly the scale factor stabilizes in practice, the companion script ch4/calibration_stability.py calibrates BERT-base layer 6 incrementally on 8, 16, 32, 64, and 128 samples, tracking the absmax-based scale factor at each step. The "% of Final" column reports each step's scale relative to the final 128-sample value, and the delta column shows the change from the previous step. Convergence is reached when both stabilize:

```
Calibration Scale Factor Convergence (BERT-base, Layer 6):
============================================================
Samples    | Scale Factor |   % of Final |  Δ from Prev | Status
------------------------------------------------------------
8          |        25.86 |        99.8% |           -- | Converg
16         |        25.90 |       100.0% |        +0.2% | Converg
32         |        25.90 |       100.0% |        +0.0% | Converg
64         |        25.90 |       100.0% |        +0.0% | Converg
128        |        25.90 |       100.0% |        +0.0% | Converg
------------------------------------------------------------
Convergence point: ~8 samples (95% of final scale factor)
```

This experiment used synthetic "diverse" texts on BERT-base—a controlled setting. The rapid convergence (8 samples!) tells us the activation ceiling was discovered early because the text variation didn't actually stress the model's activation range. In production with truly diverse real-world data, convergence typically takes longer.

The key insight isn't the specific convergence point—it's the shape of the process: early samples discover most of the range quickly, subsequent samples occasionally push the ceiling higher, and eventually new samples stop contributing new extremes.

Why 128-512 works in practice

Production calibration sets from GPTQ/AWQ authors contain genuinely diverse samples from Wikipedia articles, code repositories, multilingual text, and edge cases discovered through extensive experimentation. Their

convergence happens around 128-512 samples because that's sufficient diversity to capture real activation extremes—not because it's a magic number.

The real requirement: Your calibration set needs enough samples to discover the true activation ceiling for your deployment domain. If your 128 samples don't include the input patterns that produce maximum activations, you'll underestimate the range regardless of sample count. Diversity of activation patterns matters more than raw sample count.

Validating calibration against production

Before trusting your calibration set, verify it covers production patterns. This requires a deliberate workflow.

Deploy at full precision first when feasible — your initial production deployment runs in FP16/FP32 with activation instrumentation enabled, establishing ground truth for production. When cost or memory makes a full FP32 rollout infeasible, run a shadow FP32 replica against a sampled fraction of production traffic for 24-48 hours to capture activation statistics before quantizing the primary deployment. Collect production activation statistics by running for sufficient time to capture temporal patterns—at minimum a full day/week cycle, ideally covering seasonal variation if relevant. Compare calibration coverage against production using validation scripts.

The companion script ch4/validate_calibration.py automates this coverage check. It reads the per-layer ranges produced by your calibration observer and the production activation log captured during the FP32 shadow run, then compares them layer by layer. Each layer gets a Coverage percentage (calibration max divided by production max) and a status flag: OK when coverage is at or above 100%, WARNING below 100%, CRITICAL below 90%. The first-pass report below shows the kind of gap you'd otherwise discover only as an accuracy regression in deployed traffic:

```
Calibration Validation Report:
============================================================
Overall Status: FAIL ✗
Layers: 3 OK, 1 Warning, 1 Critical
```

```
Maximum Gap: 16.6%

Per-Layer Coverage:
------------------------------------------------------------------
Layer        |  Calib Max |   Prod Max |   Coverage | Status
------------------------------------------------------------------
layer_6      |       42.8 |       51.3 |      83.4% | CRITICAL ✗
layer_0      |       25.4 |       28.1 |      90.4% | WARNING ⚠
layer_3      |       32.1 |       31.5 |     101.9% | OK ✓
layer_9      |       38.2 |       36.8 |     103.8% | OK ✓
layer_11     |       38.5 |       36.2 |     106.4% | OK ✓
==================================================================
```

The report identifies layer_6 as critical—production activations exceed calibration by 16.6%, meaning quantized inference will clip values and degrade accuracy.

After adding samples that stress the undercovered layers:

```
Calibration Validation Report:
==================================================================
Overall Status: PASS ✓
Layers: 5 OK, 0 Warning, 0 Critical

Per-Layer Coverage:
------------------------------------------------------------------
Layer        |  Calib Max |   Prod Max |   Coverage | Status
------------------------------------------------------------------
layer_3      |       32.1 |       31.5 |     101.9% | OK ✓
layer_9      |       38.2 |       36.8 |     103.8% | OK ✓
layer_11     |       38.5 |       36.2 |     106.4% | OK ✓
layer_6      |       55.0 |       51.3 |     107.2% | OK ✓
layer_0      |       30.2 |       28.1 |     107.5% | OK ✓
==================================================================
```

All layers now show >100% coverage—calibration ranges exceed production observations, ensuring no clipping. This feedback loop is essential. Production traffic evolves—new user segments, seasonal shifts, feature changes—and calibration sets need periodic revalidation.

The mental anchor for calibration:

Calibration data defines your quantization grid. Mismatched calibration is a silent accuracy killer — the model runs, but outputs are wrong.

"Representative" means covering activation space, not input distribution; a handful of challenging inputs that stress activation ranges outweigh thousands of typical samples. Diversity beats volume — 128–512 diverse samples typically suffice, focused on linguistic/visual variety and adversarial edge cases. Validate against production: deploy full-precision first, instrument activations, then verify calibration coverage before quantizing. Revalidate periodically as traffic evolves.

4.3 Estimate ranges: absolute max, percentile, and MSE

You've collected a representative calibration set and run it through your model. The activation observers have accumulated millions of values—minimums, maximums, histograms, running statistics. Now comes the question that will determine whether your quantized model works or fails: What value should you actually use as the range boundary?

The answer seems obvious—use the observed maximum. But that obvious answer is often wrong. A single activation spike of 11.5 in a distribution where 99% of values never exceed 1.9 will stretch your scale factor by 6×, destroying precision for the vast majority of your values. This section explores four range-estimation strategies: absolute maximum (simple but outlier-sensitive), percentile clipping (robust but lossy), MSE-optimal (principled, more compute), and KL-divergence / entropy (popular in TensorRT, but a cautionary tale on heavy-tailed distributions, as we'll see). Understanding when each succeeds and fails is the difference between a model that "works" on calibration and one that ships.

Absolute maximum: the naive baseline

The absolute maximum method is quantization's "hello world." It's what you'd write if you had 30 seconds to implement range estimation:

Listing 4.3 Absolute maximum range estimation

```
def absmax_range(tensor):
```

```
    """Simplest possible range estimation."""
    return tensor.abs().max().item()
```

For symmetric quantization, this single value defines everything: the scale factor becomes *S=max_abs/127* (for INT8), and values outside [-max_abs, max_abs] would be clipped—except by definition, no such values exist.

The mathematics of zero clipping

Absolute maximum has one virtue: it guarantees zero clipping error *on the calibration set*. Every observed value fits within the quantization range by construction. For 8-bit symmetric quantization, the scale becomes S = absmax(x)/127, and the reconstruction formula $x\hat{} = S \cdot round(x/S)$ introduces only a rounding error, bounded by S/2 per value. Two caveats: production activations can still exceed the calibrated range and get clipped at inference time; and a single outlier in calibration inflates S, making the S/2 rounding error damaging for the bulk of values (bounded in magnitude, but large enough to dominate downstream errors).

The outlier problem in practice

Chapter 3 established that transformer activations exhibit large outlier ratios. To see what this costs us under absmax calibration, listing 4.4 attaches a forward hook to BERT-base layer 6's FFN output, runs diverse calibration text through the model, and reports two numbers per sample: the absolute maximum and the 99th percentile of activation magnitudes. The ratio between them — the *outlier ratio*, tells us directly how much of the INT8 range is being consumed by the long tail rather than by typical values.

Listing 4.4 Analyzing absmax limitations

```
def analyze_absmax_problem(model, tokenizer, texts):
    activations = []

    def hook(module, input, output):
        tensor = output[0] if isinstance(output, tuple) else outp
        activations.append(tensor.detach())

    handle = model.encoder.layer[6].output.register_forward_hook(
```

```
    with torch.no_grad():
        for text in texts:
            inputs = tokenizer(text, return_tensors='pt', truncat
            model(**inputs)

    handle.remove()
    all_acts = torch.cat([a.flatten() for a in activations])

    absmax = all_acts.abs().max().item()
    p99 = torch.quantile(all_acts.abs(), 0.99).item()

    return {'absmax': absmax, 'p99': p99, 'outlier_ratio': absmax
```

Running this on BERT-base with diverse calibration text:

```
Activation Statistics (BERT layer 6 FFN output, 89,088 values):
──────────────────────────────────────────────────────────────
  Absolute max:      11.52
  99th percentile:    1.92
  99.9th percentile:  6.25
  Outlier ratio:      5.99x (max / p99)
──────────────────────────────────────────────────────────────
```

The numbers tell a stark story. The 99th percentile is just 1.92, but the maximum reaches 11.52—a 6× outlier ratio. This means 1% of the dynamic range (from the outliers) is consuming precision that could serve the other 99% of values.

When absolute maximum works

Despite its limitations, absolute maximum calibration remains appropriate in specific scenarios. For weight quantization, weights are static and typically well-behaved with outlier ratios of 1.2-2×, making the precision loss acceptable. For ReLU activations in CNNs without transformer-style attention, CNN activations tend to have tighter distributions with outlier ratios commonly under 3×. For tasks with hard clipping constraints where your application absolutely cannot tolerate any clipping (certain safety-critical systems), absmax is the only guarantee. And as a quick baseline for initial sanity checks before investing in more sophisticated methods.

The insight behind percentile clipping is simple: if 0.01% of values cause 6×

precision loss for the other 99.99%, maybe those outliers should be sacrificed. Instead of using the absolute maximum, we use a percentile (typically 99th, 99.9th, or 99.99th) as our range boundary. Listing 4.5 implements this as a drop-in replacement for absmax — an observer that records every value it sees during calibration, then returns the requested percentile when asked to compute the quantization range.

Listing 4.5 Percentile-based observer

```
class PercentileObserver:
    def __init__(self, percentile=99.99):
        self.percentile = percentile
        self.all_values = []

    def observe(self, tensor):
        self.all_values.append(tensor.abs().flatten().cpu())

    def compute_range(self):
        all_abs = torch.cat(self.all_values)
        return torch.quantile(all_abs, self.percentile / 100).ite
```

The clipping-resolution trade-off

Percentile clipping accepts bounded clipping error for a subset of values in exchange for better resolution for the majority. As we derived in Chapter 2, granular (rounding) error scales with S/2, while overload (clipping) error grows with how far outliers exceed the range. The trade is favorable whenever the clipping cost for 0.01% of values is less than the rounding savings for the other 99.99%.

The optimal percentile depends on the distribution shape and the downstream task's sensitivity to errors. Table 4.1 turns this into a working starting point: each row pairs a distribution profile (clean, mildly tailed, heavy-tailed, extreme outliers) with a percentile recommendation and a representative model class, so you can pick a reasonable default rather than running a full sweep.

Table 4.1 Percentile selection guide

Percentile	Best for	Risk	Typical Use
99.99%	Conservative, minimal clipping	May still be dominated by outliers	CNN activations, general default
99.9%	Moderate outlier distributions	0.1% values clipped	Transformer FFN layers
99.0%	Heavy-tailed, severe outliers	1% values clipped, aggressive	Attention outputs, LLM activations
100% (absmax)	Zero clipping tolerance	Precision loss from outliers	Weights, safety-critical apps

The non-obvious feature of this trade-off is that the relationship between percentile threshold and total error is non-monotonic — clipping more aggressively doesn't monotonically improve or worsen MSE. The optimum is empirical; in most heavy-tailed cases it lies between 99% and 100%, but it is not guaranteed to be interior (an extremely well-behaved distribution can have its optimum at absmax itself).

Rather than guessing, we can directly measure the clipping vs. resolution trade-off. Running a percentile sweep on BERT-base layer 6 activations:

```
PERCENTILE SWEEP ANALYSIS
──────────────────────────────────────────────────────────────────────
Percentile   Range      Clipped    MSE          SNR (dB)
──────────────────────────────────────────────────────────────────────
99.00        1.9227     1.0001%    0.068384     8.95
99.50        2.2657     0.5006%    0.060052     9.52
99.90        6.2455     0.1010%    0.011418     16.73
99.95        9.7443     0.0505%    0.000833     28.09
99.99        10.7027    0.0101%    0.000604     29.49       ← BEST
100.00       11.5178    0.0000%    0.000688     28.93
──────────────────────────────────────────────────────────────────────
Optimal percentile: 99.99%
```

The data reveals a nuanced story. Aggressive clipping at the 99th percentile (clipping 1% of values) actually hurts performance — the clipping error overwhelms any resolution gains, producing an MSE of 0.0684 vs the 99.99th percentile's 0.000604 (≈113× worse). But the 99.99th percentile hits the sweet spot: clipping just 0.01% of values while achieving the lowest total

MSE, even beating absmax despite introducing some clipping.

MSE-optimal calibration: finding the sweet spot

The percentile approach works well but requires choosing the percentile value a priori. What if we could directly find the range that minimizes reconstruction error? This is the idea behind MSE-optimal calibration: search over possible ranges and select the one that minimizes mean squared error between original and quantized values.

The optimization problem

Given a tensor x and a candidate range α, the quantization-dequantization process produces x_{hat}. We seek $\alpha^* = arg\ min_\alpha\ MSE(x, \hat{x}(\alpha))$. This is a one-dimensional optimization problem, but it's not convex—the objective has discontinuities where the clipping boundary crosses data points. However, the search space is bounded: α lies between half the absolute maximum and the absolute maximum itself (the lower bound is a practical choice in the reference implementation — going below it rarely yields better MSE and risks clipping the bulk of the distribution).

Listing 4.6 implements this search efficiently—rather than evaluating MSE on the full tensor for every candidate α, we accumulate a histogram of absolute values and compute the weighted error against bin counts, reducing what would be an expensive grid search to a lightweight inner loop.

Listing 4.6 MSE-optimal observer with histogram acceleration

```
class MSEOptimalObserver:
    def __init__(self, num_bins=2048, num_candidates=200):
        self.num_bins = num_bins
        self.num_candidates = num_candidates
        self.histogram = None
        self.max_val = 0.0

    def observe(self, tensor):
        abs_vals = tensor.abs().float()
        curr_max = abs_vals.max().item()
        if curr_max > self.max_val:
```

```
            self.max_val = curr_max

        hist = torch.histc(abs_vals.flatten(), bins=self.num_bins
                           min=0, max=self.max_val)
        if self.histogram is None:
            self.histogram = hist.cpu()
        else:
            self.histogram += hist.cpu()

    def compute_range(self):
        candidates = torch.linspace(0.5 * self.max_val, self.max_
                                    self.num_candidates)
        bin_width = self.max_val / self.num_bins
        bin_centers = torch.arange(self.num_bins) * bin_width + b

        best_mse, best_range = float('inf'), self.max_val

        for r in candidates:
            scale = r / 127
            clipped = torch.clamp(bin_centers, 0, r.item())
            quantized = torch.round(clipped / scale) * scale
            squared_error = (bin_centers - quantized) ** 2
            mse = (squared_error * self.histogram).sum() / self.h

            if mse < best_mse:
                best_mse, best_range = mse, r.item()

        return best_range
```

On BERT-base, MSE-optimal calibration finds a range of 10.59—slightly tighter than the 99.99th percentile (10.70) but not as aggressive as lower percentiles. This results in the lowest MSE of all methods tested:

```
CALIBRATION RESULTS
```

Method	Range	Scale	Clipped	MSE	S
mse	10.5917	0.083399	0.0236%	0.000600	2
percentile	10.7027	0.084273	0.0101%	0.000604	2
absmax	11.5178	0.090691	0.0000%	0.000688	2
entropy	0.7199	0.005668	24.9989%	0.160087	5

```
Best method: mse (MSE: 0.000600)
```

A practical note on histogram-based search: the result depends on binning resolution and candidate density. Too few bins (say 64) smear the distribution

and hide narrow peaks, biasing the search toward wider ranges. Too few candidates miss the true minimum if the MSE surface has sharp valleys. The defaults in Listing 4.6—2048 bins and 200 candidates—work well for typical activations, but layers with bimodal or very heavy-tailed distributions may need 4096+ bins. If you suspect the search found a local minimum, run it twice with different candidate counts and compare; if the results diverge by more than 1%, increase resolution. The computational cost is negligible—the histogram is accumulated once during calibration, and the grid search over candidates takes milliseconds.

KL divergence: an alternative objective

MSE minimizes reconstruction error in value space. An alternative approach —popularized by TensorRT—minimizes the information loss measured by Kullback-Leibler divergence. The intuition: we want the quantized distribution to preserve the "shape" of the original distribution, not just minimize point-wise error.

The entropy calibration failure mode

Our experimental results reveal a critical failure mode of entropy calibration on this distribution. The entropy method selected a range of just 0.72—clipping nearly 25% of all values! This catastrophic result (267× worse MSE than MSE-optimal) occurs because KL-divergence is optimizing for distribution shape preservation, not reconstruction accuracy. When the distribution has a very tall, narrow peak near zero (as BERT activations do), entropy calibration can aggressively clip the tails to better represent the peak's shape.

This illustrates a crucial point: entropy calibration is not universally applicable. It works well for distributions that are roughly uniform or Gaussian, but fails dramatically on heavy-tailed distributions common in transformers. The intuition: KL-divergence measures how well the quantized distribution approximates the original as a probability distribution. A tall peak concentrates most of the probability mass near zero—so any quantization grid that spreads its levels across a wide range wastes most of those levels on the sparse tails, producing a poor approximation of the peak.

The optimizer responds by shrinking the range until the grid is dense enough to faithfully reproduce that peak, sacrificing the tails entirely. It's optimizing the right objective—distribution shape—but that objective doesn't penalize value-level reconstruction error on the clipped outliers.

NOTE

Don't reach for entropy/KL calibration on transformer activations. The TensorRT default exists for the kinds of CNN activations TensorRT was originally designed for (roughly Gaussian, no extreme tails). Heavy-tailed distributions, including most transformer activations, cause the optimizer to collapse the range catastrophically. Always compute the outlier ratio first; if it exceeds ~3×, prefer MSE-optimal.

When MSE vs. KL divergence matters

Neither calibration method dominates across all scenarios. MSE-optimal directly minimizes reconstruction error in value space and handles heavy-tailed transformer activations gracefully. KL-divergence treats the problem as information-theoretic and works well on the roughly Gaussian distributions that TensorRT was originally tuned for. Table 4.2 lays out the choice axes side by side: the kind of distribution each method handles best, the compute cost of the search, and the failure mode each one is prone to when applied outside its sweet spot.

Table 4.2 MSE vs KL-divergence comparison

Aspect	MSE-Optimal	KL-Divergence (Entropy)
Objective	Minimize point-wise reconstruction error	Minimize information loss
Strength	Best for regression-like tasks	Best for classification when distribution is well-behaved
Weakness	May over-fit to outliers if important	Can catastrophically fail on heavy-tailed distributions
Compute Cost	Moderate (grid search)	Higher (histogram manipulation)

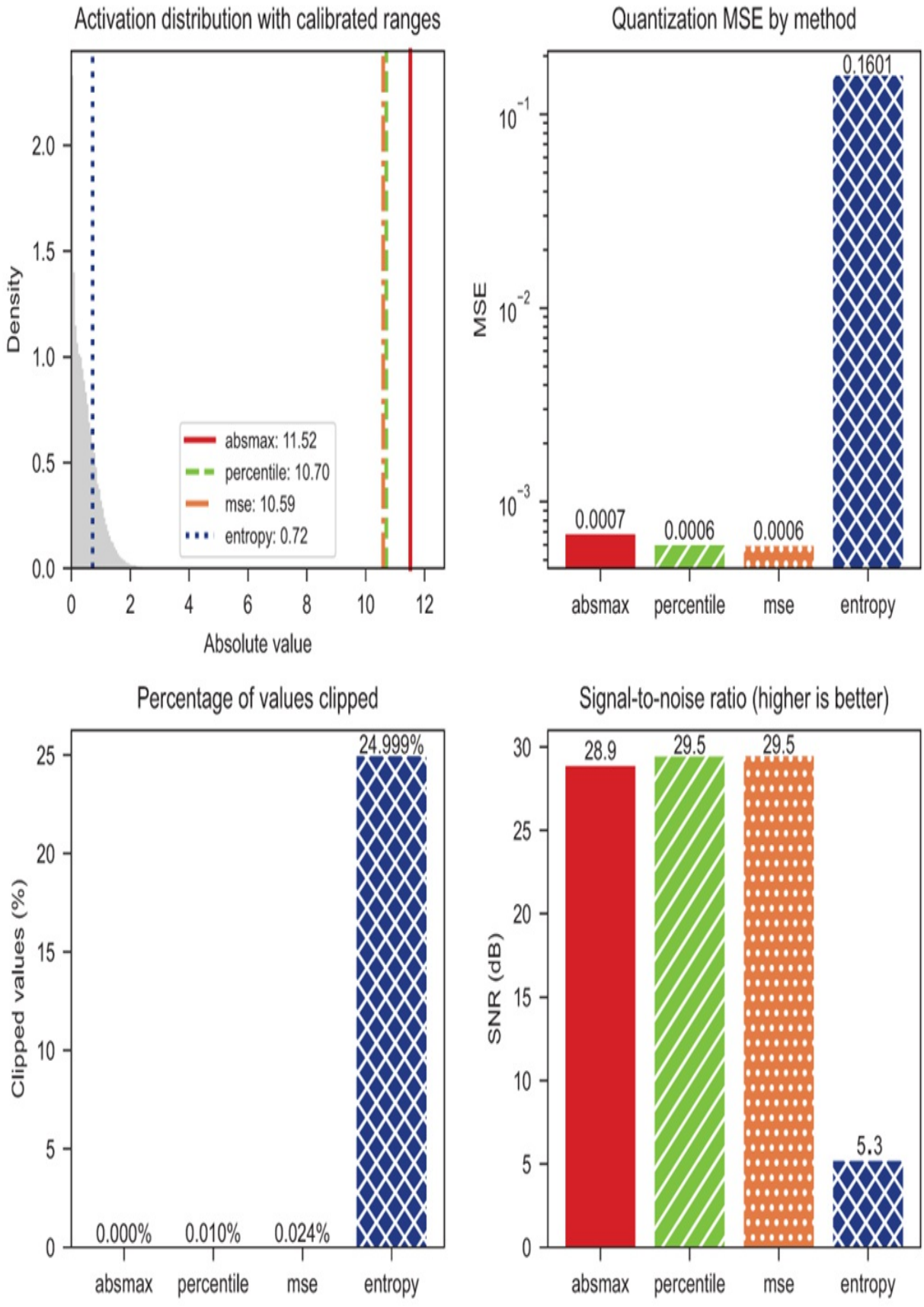
Framework	PyTorch, ONNX Runtime	TensorRT, TFLite
Recommendation	Default choice for transformers	Use with caution; verify on your distribution

Putting it together: a unified calibration framework

We've explored four methods with different trade-offs.

Figure 4.5 shows why no single method wins universally—absmax preserves the full range but wastes resolution on outliers, percentile and MSE find nearly identical sweet spots for this layer, and entropy collapses too aggressively.

Figure 4.5 Comparison of range estimation methods on BERT-base layer 6 activations


Activation distribution with calibrated ranges
Density
Absolute value
absmax: 11.52
percentile: 10.70
mse: 10.59
entropy: 0.72
Quantization MSE by method
MSE
0.1601
0.0007
0.0006
0.0006
absmax
percentile
mse
entropy
Percentage of values clipped
Clipped values (%)
24.999%
0.000%
0.010%
0.024%
absmax
percentile
mse
entropy
Signal-to-noise ratio (higher is better)
SNR (dB)
28.9
29.5
29.5
5.3
absmax
percentile
mse
entropy

In practice, you'll want to try multiple methods and compare, which means your calibration code shouldn't hardcode a single strategy. Listing 4.8 wraps all four methods behind a common interface so you can swap algorithms with a single parameter change.

Listing 4.7 Unified calibration observer

```
class UnifiedCalibrationObserver:
    METHODS = ['absmax', 'percentile', 'mse', 'entropy']

    def __init__(self, method='percentile', percentile=99.99, num
        self.method = method
        self.percentile = percentile
        self.num_bins = num_bins
        self.min_val, self.max_val = float('inf'), float('-inf')
        self.histogram = None
        self.all_values = []

    def observe(self, x):
        abs_x = x.abs().float()
        self.min_val = min(self.min_val, x.min().item())
        self.max_val = max(self.max_val, abs_x.max().item())

        if self.method in ['mse', 'entropy']:
            self._update_histogram(abs_x)
        if self.method == 'percentile':
            self.all_values.append(abs_x.flatten().cpu())

    def compute_qparams(self):
        range_val = self._compute_range()
        scale = range_val / 127
        return scale, 0
```

The diagnostic number: outlier ratio

Before choosing a method, always compute the outlier ratio:

Listing 4.8 Outlier ratio diagnostic

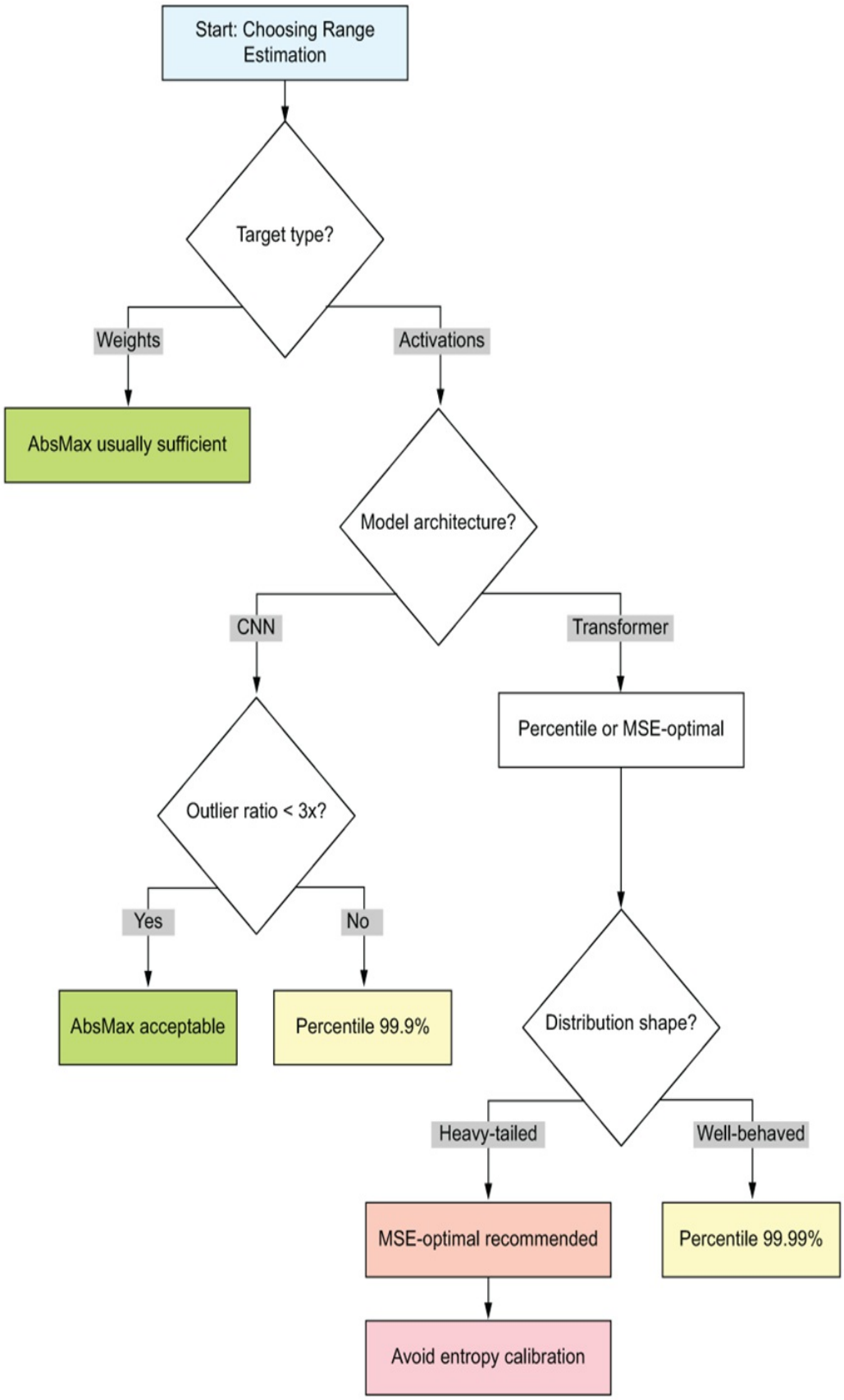
```
def compute_outlier_ratio(activations):
    """The single most predictive diagnostic for range estimation
    abs_vals = activations.abs()
```

```
    return abs_vals.max().item() / torch.quantile(abs_vals, 0.99)
```

For BERT-base layer 6, this returns 5.99×. The interpretation: outlier ratio under 3× means AbsMax is acceptable and all methods produce similar results; ratio between 3-10× means percentile (99.99%) or MSE-optimal is recommended. Ratio over 10× calls for MSE-optimal or aggressive percentile (99.0-99.9%).

Figure 4.6 codifies these thresholds into a decision framework you can follow mechanically—start with the outlier ratio, branch on the result, and arrive at a recommended method without guesswork.

Figure 4.6 Decision framework for range estimation method selection. Start by computing the outlier ratio (max / 99th percentile) on your calibration data. If the ratio is below 2×, activations are well-behaved and absolute maximum (absmax) is sufficient. Between 2–10×, percentile clipping at 99.99% provides a safe default. Above 10×, use MSE-optimal calibration to automatically find the best clipping threshold. The framework routes you to the simplest method that handles your distribution shape.


Start: Choosing Range Estimation
Target type?
Weights
Activations
AbsMax usually sufficient
Model architecture?
CNN
Transformer
Outlier ratio < 3x?
Percentile or MSE-optimal
Yes
No
AbsMax acceptable
Percentile 99.9%
Distribution shape?
Heavy-tailed
Well-behaved
MSE-optimal recommended
Percentile 99.99%
Avoid entropy calibration

The mental anchor for range estimation:

Range estimation is where calibration theory meets calibration practice. The method you choose determines how your quantization budget is allocated between clipping error and rounding error.

4.4 Validate accuracy and size with a repeatable protocol

You've built a calibration set, chosen a range estimation method, and quantized your model. Now comes the moment of truth: does it actually work?

NOTE

This section validates accuracy and size. It does not measure latency — our experiments dequantize to FP32 on every forward pass, which is correct for accuracy validation but slower than FP32. Latency measurements require runtime-specific setup (PyTorch backends, TensorRT, ONNX Runtime) and are covered in Chapter 9.

This section establishes a validation protocol that answers three questions: Does accuracy hold, and how much did we lose? Did size shrink, and is the 4× compression real? Which calibration method won, and does theory match practice?

We'll validate on two production-relevant models: ResNet-18 on ImageNet (vision) and BERT-base on SST-2 sentiment (NLP). These represent the architecture families underlying many real deployments; the same patterns extend to larger models, though absolute numbers shift.

The validation experiment

Our protocol is simple: measure FP32 baseline, quantize with each calibration method from section 4.3, and compare. The companion script

ch4_ptq_validation.py runs the full experiment:

```
python ch4_ptq_validation.py --model all --num-samples 500
```

Before running the protocol: If your model has batch normalization layers, fold them into the preceding linear/conv layers *before* calibration. Folding after calibration is one of the most common causes of unexplained accuracy regressions — the folded weights have different scale statistics than the unfolded ones the calibrator saw.

Step 1: Establish FP32 baseline

Before quantizing anything, we need ground truth. For ResNet-18 on 500 ImageNet validation samples:

```
[3/4] Evaluating FP32 baseline...
  Accuracy: 72.20%
  Size: 44.59 MB
```

The above 72.2% accuracy is close to ResNet-18's published top-1 of ~69.8% (variance from our sample). The 44.59 MB matches 11.7M parameters × 4 bytes per FP32 weight.

Step 2: Quantize and measure

We apply INT8 quantization with three calibration methods: absmax, percentile (99.99%), and MSE-optimal. Each produces identical size reduction but different accuracy:

```
ABSMAX:
    Quantization MSE: 6.69e-07
    Accuracy: 71.60% (-0.60%)
    Size: 11.18 MB (4.0x smaller)

PERCENTILE:
    Quantization MSE: 4.40e-07
    Accuracy: 70.60% (-1.60%)
    Size: 11.18 MB (4.0x smaller)

MSE:
```

```
    Quantization MSE: 3.52e-07
    Accuracy: 72.00% (-0.20%)
    Size: 11.18 MB (4.0x smaller)
```

Three observations jump out. Size reduction is guaranteed—all methods achieve exactly 4× compression because INT8 uses 1 byte where FP32 uses 4. Calibration method matters for accuracy—MSE-optimal loses only 0.20% while percentile loses 1.60%, an 8× difference in accuracy degradation from the same bit-width. Lower quantization error correlates with higher accuracy—MSE-optimal has the lowest reconstruction error (3.52e-07) and the best accuracy, validating our intuition from section 4.3.

Vision vs NLP: different models, different behaviors

BERT-base tells a different story:

```
[2/4] Evaluating FP32 baseline...
  Accuracy: 93.40%
  Size: 326.72 MB

[3/4] Analyzing calibration methods...
  absmax: MSE = 2.93e-05
  percentile: MSE = 5.50e-06
  mse: MSE = 8.26e-06

[4/4] Quantizing with best method (MSE)...
  Accuracy: 93.40% (+0.00%)
  Size: 82.03 MB (4.0x smaller)
```

BERT shows zero accuracy loss with MSE calibration. Why does BERT tolerate quantization better than ResNet? Weight distribution differences play a role—BERT's weights are predominantly in Linear layers with relatively smooth distributions, while ResNet has convolutional layers with more varied weight patterns across spatial dimensions. The calibration error numbers reveal this: BERT's MSE (8.26e-06) is an order of magnitude higher than ResNet's (3.52e-07), yet BERT's accuracy is unaffected.

This tells us something important: quantization error and accuracy loss aren't perfectly correlated across architectures. BERT's task (binary sentiment classification) has more headroom than ImageNet's 1000-way classification. A small perturbation to BERT's logits rarely flips a binary decision; the same

perturbation in ResNet might push the correct class from rank 1 to rank 2.

Table 4.3 puts concrete numbers to this intuition—we apply our best calibration methods to both architectures and measure accuracy, size, and compression side by side.

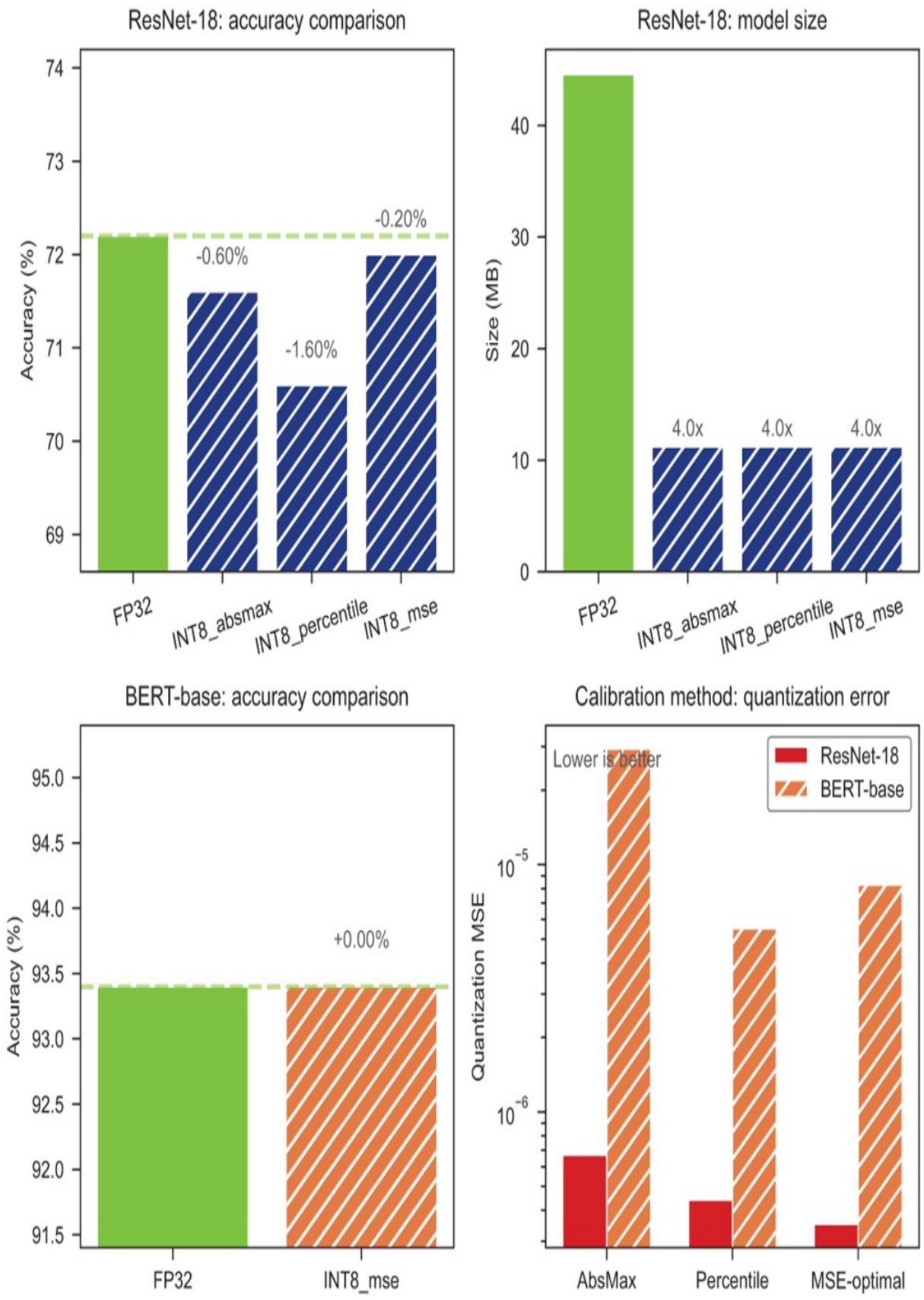
Table 4.3 PTQ validation results

Model	Config	Accuracy	Δ Accuracy	Size (MB)	Compression
ResNet-18	FP32	72.20%	—	44.59	—
	INT8 (absmax)	71.60%	-0.60%	11.18	4.0×
	INT8 (percentile)	70.60%	-1.60%	11.18	4.0×
	INT8 (mse)	72.00%	-0.20%	11.18	4.0×
BERT-base	FP32	93.40%	—	326.72	—
	INT8 (mse)	93.40%	+0.00%	82.03	4.0×

The compression ratio is identical across methods (4.0× for both models), so the only differentiator is which range estimation algorithm best preserves accuracy.

Figure 4.7 visualizes these trade-offs—accuracy retention on one axis, quantization error on the other—making the MSE-optimal advantage visible at a glance.

Figure 4.7 MSE calibration preserves accuracy best (top-left); MSE-optimal achieves lowest quantization error across architectures (bottom-right)


ResNet-18: accuracy comparison
Accuracy (%)
74
73
72
71
70
69
-0.20%
-0.60%
-1.60%
FP32
INT8_absmax
INT8_percentile
INT8_mse
ResNet-18: model size
Size (MB)
40
30
20
10
0
4.0x
4.0x
4.0x
FP32
INT8_absmax
INT8_percentile
INT8_mse
BERT-base: accuracy comparison
Accuracy (%)
95.0
94.5
94.0
93.5
93.0
92.5
92.0
91.5
+0.00%
FP32
INT8_mse
Calibration method: quantization error
Lower is better
ResNet-18
BERT-base
Quantization MSE
10^{-5}
10^{-6}
AbsMax
Percentile
MSE-optimal

But what about latency?

You might notice we haven't measured inference speed. This is intentional. Our experiment stores weights as INT8 but computes in FP32—we dequantize on every forward pass. This validates that the quantized weights produce correct outputs, but it won't show speedup. In fact, the dequantization overhead makes it slightly slower.

Real INT8 speedup requires runtime support: PyTorch with torch.ao.quantization using FBGEMM (x86) or QNNPACK (ARM) backends, TensorRT for native INT8 execution on NVIDIA GPUs, ROCm/MIGraphX for AMD Instinct GPUs, or ONNX Runtime with quantized operators and hardware acceleration. Chapter 6 covers deploying quantized models with these runtimes to achieve actual 2-4× latency improvements. For now, we've validated the harder part: INT8 weights preserve accuracy. The speedup is the *primary motivation* for most deployments and depends on runtime support — covered in Chapter 9. Two practical cautions when you do switch to a real INT8 runtime: verify operators actually execute in INT8 rather than silently upcasting to FP32, and consider keeping embedding layers and the final classification head in FP16, since both map directly to discrete outputs and tend to be disproportionately sensitive to quantization.

The mental anchor for validating accuracy:

PTQ validation answers a simple question: did quantization work? Size reduction is guaranteed — INT8 gives exactly 4× compression over FP32. Accuracy depends on calibration — MSE-optimal preserved ResNet accuracy within 0.2% while percentile lost 1.6%. Architecture matters — BERT showed zero accuracy loss while ResNet was more sensitive. And latency requires runtime support — storing INT8 weights is necessary but not sufficient for speedup. Above all, quantization cannot rescue accuracy the model never had — if FP32 is weak on a slice, INT8 will be weaker. Validate on your hardest slices, not just aggregate metrics. The validation script provides a template for your own models; run it before every deployment, and you'll catch quantization failures before your users do.

4.5 Summary

- PTQ works best for INT8 quantization of CNNs and BERT-scale transformers with smooth weight distributions—it delivers 4× size reduction with minimal accuracy loss and zero retraining.
- Calibration data determines quantization ranges, so your calibration set must match production data distribution. A few hundred representative samples usually suffice, covering edge cases and deployment scenarios.
- MSE-optimal range estimation delivered the best accuracy in our experiments (ResNet: -0.20%, BERT: 0.00% loss), though absmax works for clean distributions and percentile helps with sparse outliers.
- Size reduction is guaranteed (4× for INT8 is arithmetic), but accuracy preservation depends on calibration quality—always validate against FP32 baseline before deployment.
- Latency improvement requires runtime support (PyTorch quantization backends, TensorRT, or ONNX Runtime)—storing INT8 weights alone won't speed up inference without proper execution backends.
- When PTQ fails to meet accuracy targets, the path forward is Quantization-Aware Training (Chapter 5) for strict requirements or specialized LLM techniques (Chapter 7) for sub-8-bit precision.

5 Recovering Accuracy with Quantization-Aware Training

This chapter covers

- Recognizing when post-training quantization falls short
- Training with fake quantization and straight-through estimators
- Scheduling QAT to minimize training cost
- Per-channel strategies for convolution and linear layers
- Adapting transformers efficiently

Chapter 4 established post-training quantization as the practitioner's first choice—fast, cheap, and effective for the majority of INT8 deployments. For ResNet-18 and BERT-base at INT8, PTQ delivered near-lossless compression with zero retraining.

But PTQ has limits. When you push to INT4, when you quantize small models with little redundancy, when your accuracy budget is measured in tenths of a percent—PTQ's "observe and freeze" approach runs out of runway. The model has learned nothing about operating in low precision; it simply endures whatever distortion quantization imposes.

Quantization-aware training (QAT) changes the contract. Instead of hoping the model tolerates quantization, we teach it to expect quantization. We insert fake quantization operators into the forward pass during training, let gradients flow through them, and allow the model to adapt its weights to minimize loss in the quantized regime. The weights learn to position themselves favorably on the integer grid, avoiding regions where rounding error accumulates.

The goal is surgical precision: apply QAT where PTQ falls short, withdraw it when it's no longer needed, and never pay for retraining that a simpler intervention could have avoided. That requires knowing when QAT is warranted, how to implement it efficiently, and where to stop.

5.1 Recognize when post-training methods fall short

We previously presented PTQ as the default approach, with QAT reserved for cases where PTQ fails. But what does "failure" actually look like? The answer isn't always a dramatic accuracy collapse—sometimes it's a subtle degradation that compounds across tasks, or a specific failure mode on edge cases that your validation set didn't catch.

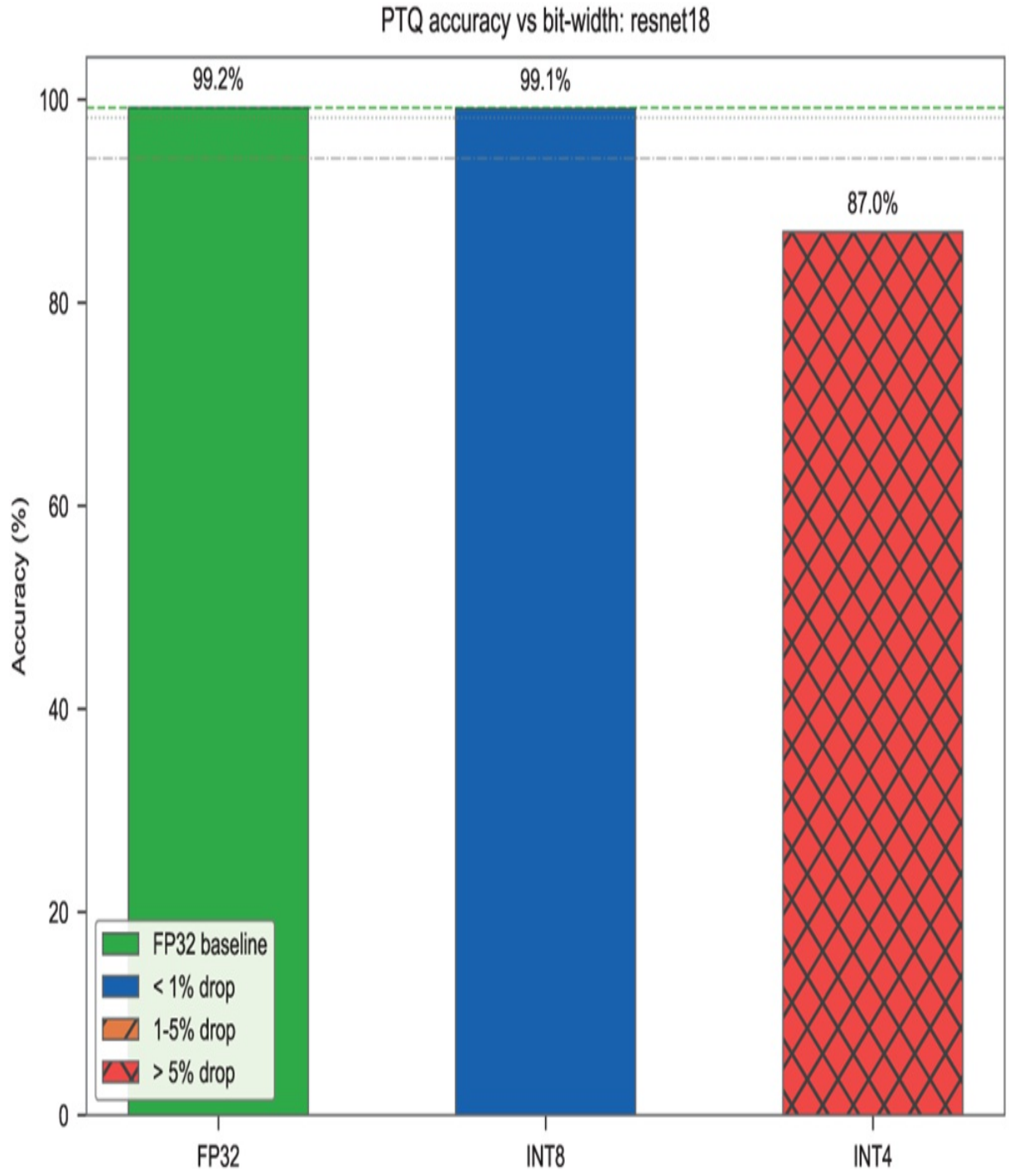
We'll examine the failure signatures across bit-widths and architectures, then develop the judgment to know when QAT's additional cost is justified.

5.1.1 The accuracy cliff

PTQ's limitations emerge gradually as you stress the quantization system along three axes: lower bit-width, smaller model size, and tighter accuracy requirements. Understanding where the cliff appears for your specific deployment lets you choose the right tool.

Figure 5.1 shows this cliff for ResNet-18 on ImageNette, comparing FP32, INT8, and INT4 PTQ accuracy side by side.

Figure 5.1 PTQ accuracy for ResNet-18 on ImageNette. INT8 is indistinguishable from the FP32 baseline. INT4 drops 12 percentage points—the model is degraded but still functioning.



The visual tells the story immediately. The FP32 baseline at 99.2% holds steady through INT8, touching the 0.1% degradation threshold. But INT4 drops well below the 5% degradation line into marginal territory.

Listing 5.1 implements this sweep, quantizing the same pretrained ResNet-18 at INT8 and INT4 and reporting the accuracy delta from the FP32 baseline.

Listing 5.1 PTQ accuracy sweep across bit-widths

```
def run_bitwidth_sweep(model, data_loader, device, eval_fn,
                       bits_list=[8, 4]):
    """Sweep across bit-widths and measure accuracy degradation."
    fp32_acc = eval_fn(model, data_loader, device)
    print(f"FP32 baseline: {fp32_acc:.2f}%")

    results = {'fp32': fp32_acc}
    for bits in bits_list:
        config = QuantConfig(bits=bits, per_channel=True)
        q_model = apply_ptq_to_model(model, config)  #A
        acc = eval_fn(q_model, data_loader, device)
        delta = acc - fp32_acc
        results[f'int{bits}'] = acc
        print(f"INT{bits}: {acc:.2f}% (Δ = {delta:+.2f}%)")

    return results
```

```
PTQ Accuracy by Bit-Width (ResNet-18, ImageNette):
```

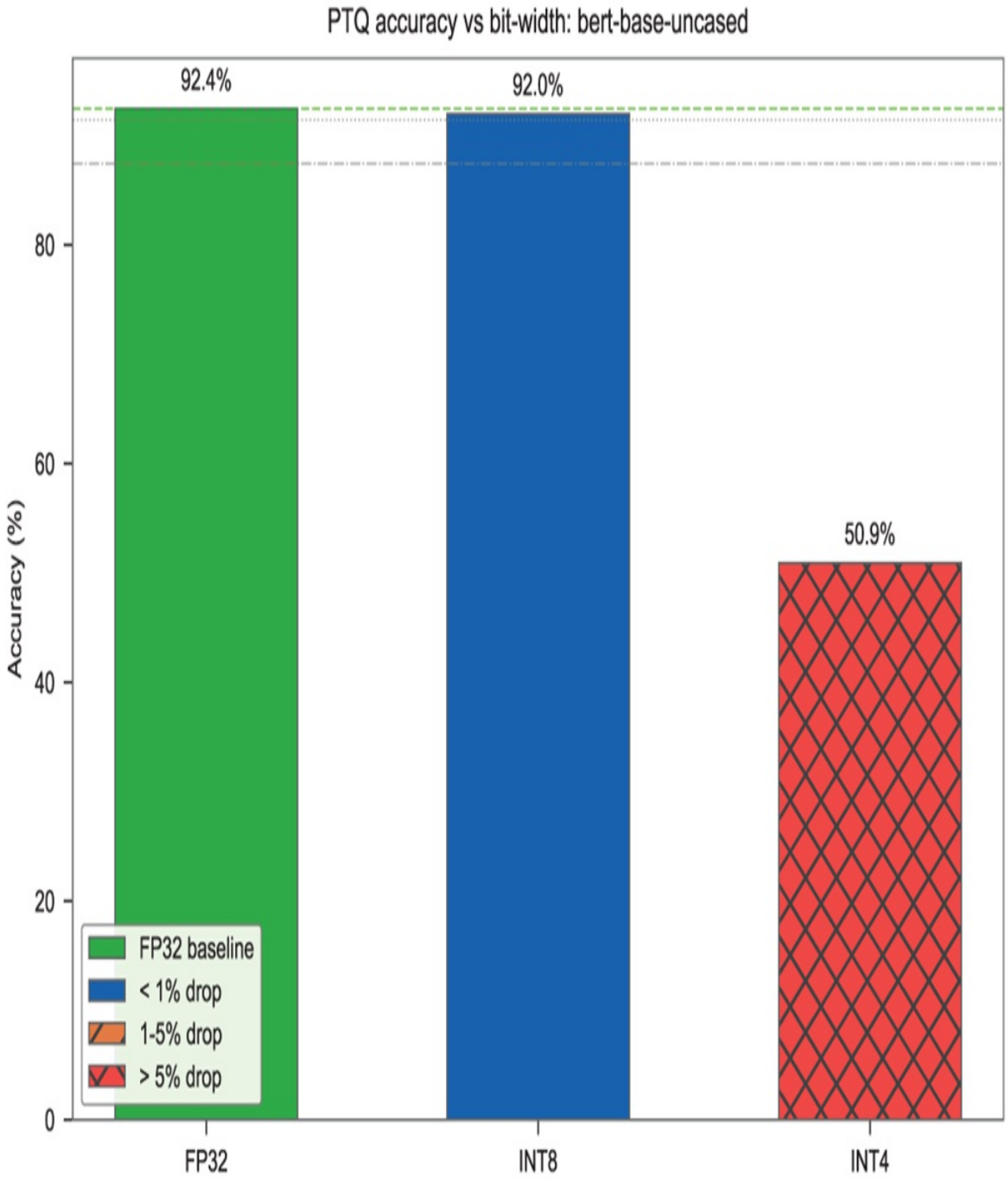
Bits	Accuracy	Δ from FP32	Status
FP32	99.21%	—	Baseline
INT8	99.11%	-0.10%	✓ Acceptable
INT4	86.98%	-12.23%	✗ Severe degradation

ResNet-18 at INT4 drops from 99.2% to 87.0%—a 12-point loss. The model is clearly damaged, but it's still a functioning classifier: 87% is well above random chance (10% for ImageNette's 10 classes). This is the regime where intervention is most valuable. The model has enough remaining signal that QAT has something to work with.

Now compare with BERT on binary sentiment classification. Figure 5.2 runs the same bit-width sweep on BERT-base with SST-2, and the result is far worse.

Figure 5.2 PTQ accuracy for BERT-base on SST-2. INT8 holds with under 0.5% loss. INT4 drops to 50.9%—coin-flip accuracy on a binary task, meaning the model has effectively stopped

functioning.



PTQ Accuracy by Bit-Width (BERT-base, SST-2):

Bits	Accuracy	Δ from FP32	Status

```
FP32  │  92.43%  │     —       │ Baseline
INT8  │  91.97%  │   -0.46%    │ ✓ Acceptable
INT4  │  50.92%  │  -41.51%    │ ✗ Catastrophic
```

BERT at 50.9% on a binary task is not a degraded model—it's a dead one. A coin flip gives you 50%. The model has lost all learned signal.

The contrast is striking: the smaller model (ResNet-18, 11.7M parameters) survives INT4 better than the larger one (BERT, 109M parameters). This is counterintuitive—you'd expect the larger model's redundancy to absorb more quantization noise. The explanation lies not in model size but in *how* each architecture responds to quantization, which the layer sensitivity analysis will reveal shortly.

5.1.2 Diagnostic signatures of PTQ failure

Knowing that INT4 hurts is the starting point. The next question is: *how* does the model fail? Different failure signatures point toward different interventions—and as we'll see, ResNet-18 and BERT fail in fundamentally different ways, demanding fundamentally different solutions.

Signature 1: Layer sensitivity patterns

Some layers are far more sensitive to quantization than others. Identifying whether sensitivity is concentrated in a few hot spots or diffuse across the network determines whether mixed precision (keeping sensitive layers at higher bit-width) can help, or whether you need QAT to retrain the whole model.

The procedure is straightforward: quantize one layer at a time (weights and activations), measure the accuracy drop, and rank layers by sensitivity.

Listing 5.2 implements this one-at-a-time perturbation analysis.

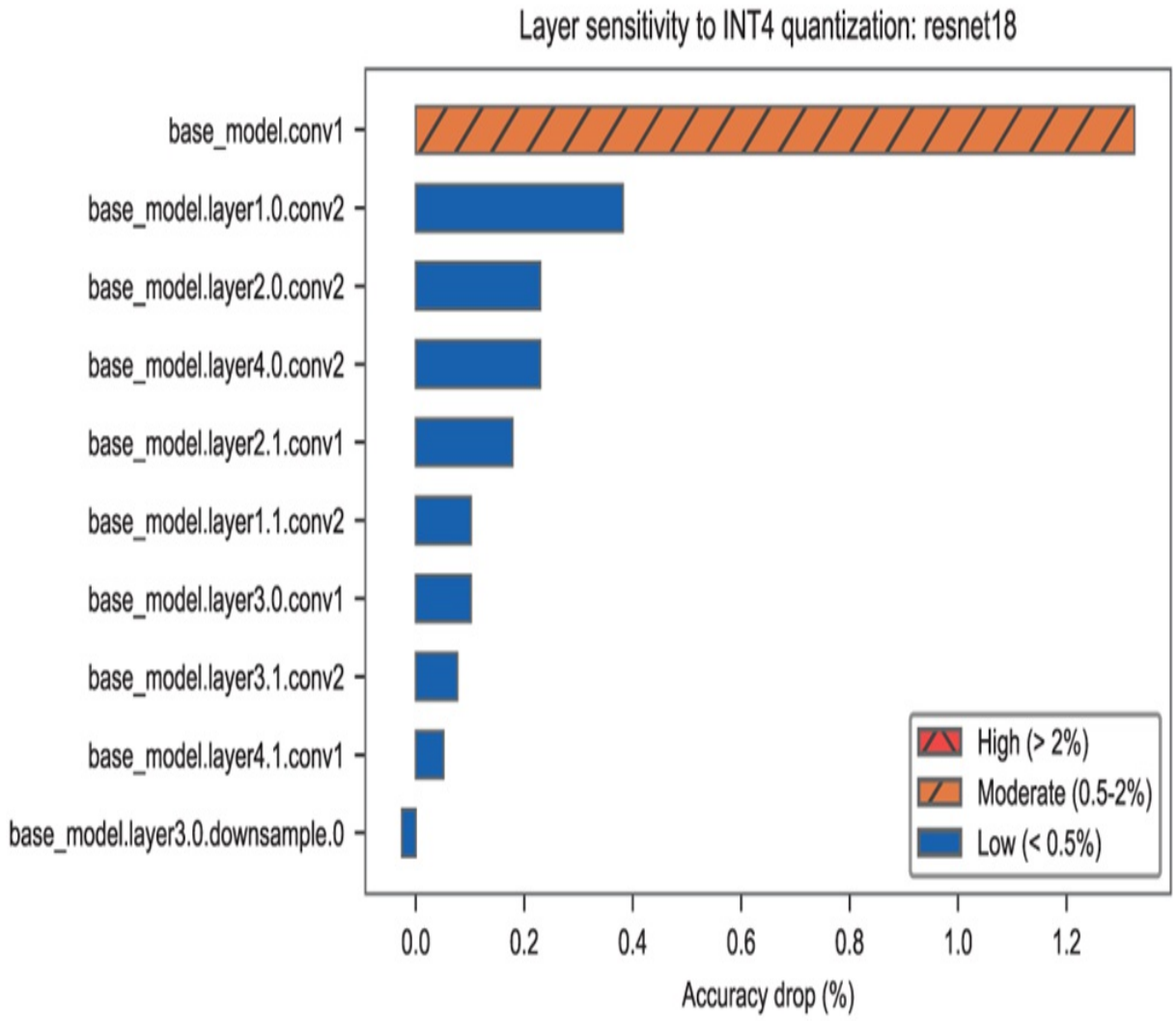
Listing 5.2 Layer sensitivity analysis

```
def run_layer_sensitivity_analysis(model, data_loader, device, ev
    """Quantize one layer at a time and measure impact."""
```

```
    baseline_acc = eval_fn(model, data_loader, device)
    sensitivities = {}

    for name, module in model.named_modules():
        if isinstance(module, (nn.Linear, nn.Conv2d)):
            # Quantize only this layer (weights + activations)
            q_model = quantize_single_layer(model, name, bits=bit
            acc = eval_fn(q_model, data_loader, device)
            sensitivities[name] = baseline_acc - acc  #A

    return sorted(sensitivities.items(), key=lambda x: x[1], reve
```

Running this analysis on ResNet-18 at INT4 produces the ranking in Figure 5.3. The key question is whether sensitivity concentrates in a few layers or spreads evenly across the network—because concentrated failure can be fixed with mixed precision, while diffuse failure requires QAT.

Figure 5.3 Layer sensitivity analysis for ResNet-18 at INT4. The first convolutional layer (conv1) shows the highest sensitivity at 1.32%, followed by a smooth gradient across the remaining layers. All bars are moderate or low—no single layer dominates the 12% total degradation.



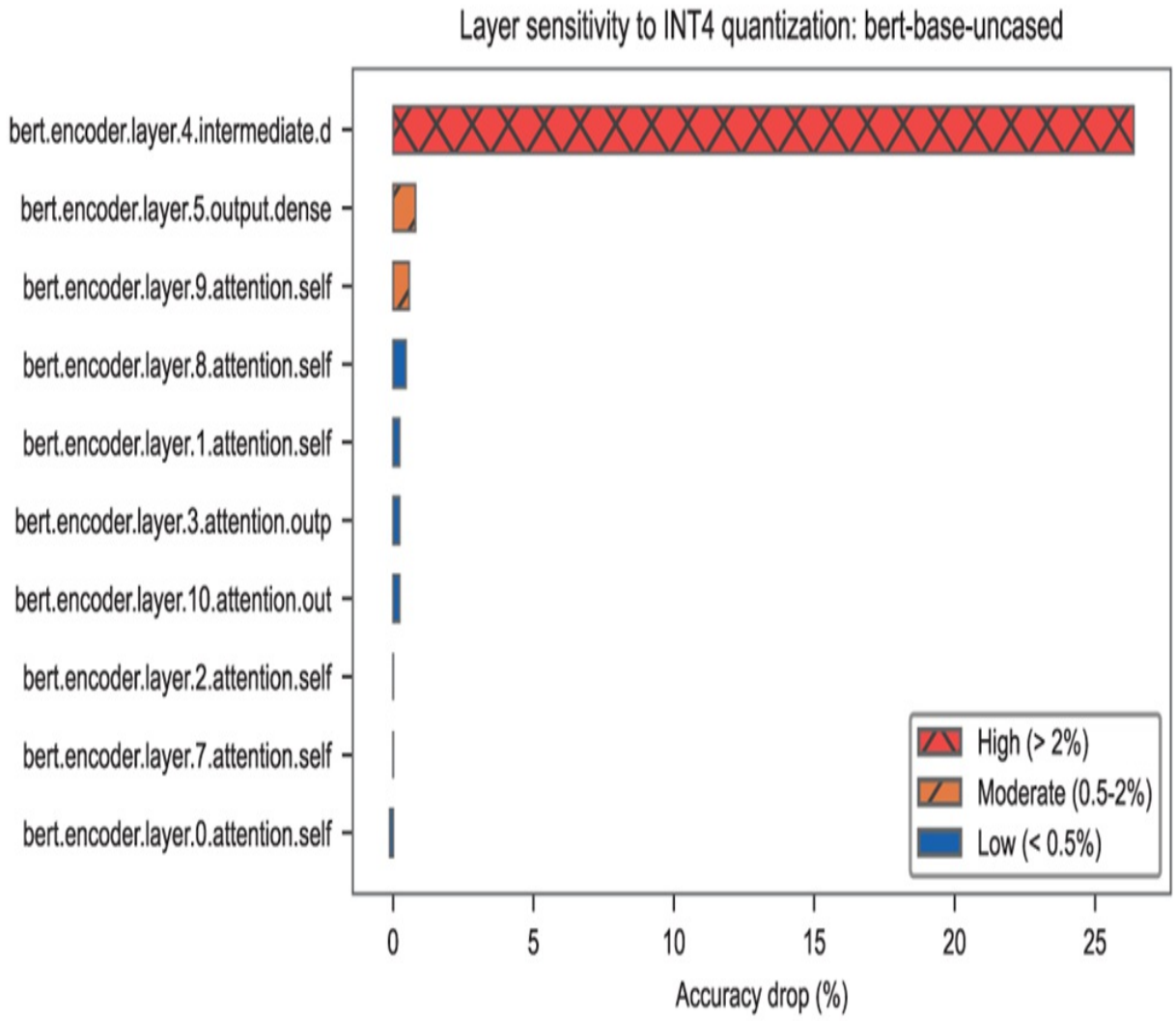
Layer Sensitivity Analysis (ResNet-18 @ INT4):

Layer	Accuracy Drop	Status
base_model.conv1	1.32%	Moderate
base_model.layer1.0.conv2	0.38%	Low
base_model.layer2.0.conv2	0.23%	Low
base_model.layer4.0.conv2	0.23%	Low
base_model.layer2.1.conv1	0.18%	Low
base_model.layer1.1.conv2	0.10%	Low
base_model.layer3.0.conv1	0.10%	Low
base_model.layer3.1.conv2	0.08%	Negligible
base_model.layer4.1.conv1	0.05%	Negligible
base_model.layer3.0.downsample	-0.03%	Negligible

The most sensitive layer (`conv1`) causes only 1.32% degradation when quantized alone—about 11% of the total 12.2% loss. Summing all ten sampled layers' individual sensitivities gives about 2.6%, yet quantizing them all simultaneously costs 12.2%. The gap between individual and joint degradation tells you the damage comes from error *compounding* across layers, not from any single hot spot. This is a diffuse failure pattern: mixed precision won't help because there's no bottleneck to target.

Now compare again with BERT. The story in figure 5.4 is completely different.

Figure 5.4 Layer sensitivity analysis for BERT at INT4. One layer dominates everything: layer.4.intermediate.dense causes a 26.4% accuracy drop when quantized alone. The remaining layers are noise in comparison. This is a concentrated failure—a clear target for mixed precision.



One layer - the FFN up-projection of encoder block 4 - accounts for 26.4% of accuracy loss alone, nearly two-thirds of the total 41.5% degradation. Why this specific layer? Transformer FFN intermediate layers apply a nonlinear expansion (typically 4× the hidden dimension), and their activations develop extreme outlier values after layer normalization. At INT4's 16 levels, these outliers get clipped or severely distorted, destroying the signal that downstream layers depend on.

The practical consequence: *for BERT, you likely don't need QAT at all.* Keeping layer.4.intermediate.dense in INT8 while quantizing everything else to INT4 should recover most of the lost accuracy - no retraining, just a configuration change.

The contrast is exactly why running layer sensitivity analysis matters before reaching for QAT. Without it, you'd apply the same heavy-handed solution to both models when one needs a scalpel and the other needs a sledgehammer.

Signature 2: Per-class accuracy degradation

Layer sensitivity tells you *where* in the model the damage occurs. Per-class analysis tells you *what* the model gets wrong—and can reveal pathological failure modes that aggregate accuracy numbers hide.

When we break down ResNet-18's INT4 performance by ImageNette class, the picture is degraded but rational. Every class loses accuracy—the worst three (hard disk drive, chain saw, garbage truck) drop over 20 percentage points from their FP32 baselines, falling from the high 90s into the mid-to-high 70s. The two best classes still clear 94%. Crucially, no class collapses entirely. The damage is non-uniform but universal: a mean degradation of 12.2% with a standard deviation of 6.0% across ten classes. This matches the diffuse layer sensitivity we just saw—many small injuries across the network, no single fatal blow. The model needs a global intervention (QAT) rather than a targeted one.

BERT tells a completely different story. The quantized model predicts class 1 (positive sentiment) for *every single input*. Class 0 (negative) accuracy drops from 91.4% to 0.0%; class 1 jumps to 100.0%. This is not degradation—it's mode collapse. The overall 50.9% accuracy is simply the base rate of positive examples in the SST-2 validation set. The model hasn't become noisy; it's become trivial—and this kind of failure is invisible to aggregate metrics. A mode-collapsed model is maximally confident in its wrong predictions for an entire class.

(The companion script ch5/ch5_ptq_failure_diagnostics.py `--per-class` *flag runs this full analysis on your own models.)*

Signature 3: Confidence distribution shift

Even when a model maintains reasonable top-1 accuracy, its confidence distribution may have shifted in ways that signal fragility—or, in BERT's

case, confirm the mode collapse just uncovered.

For each prediction, we record the model's maximum softmax probability—its confidence in its top choice—and compare FP32 against INT4. ResNet-18 at FP32 is a confident classifier: mean confidence of 0.992, with only 0.1% of predictions falling below 50% confidence. INT4 pushes that low-confidence tail from 0.1% to 2.5%—a 25× increase. Most predictions remain confident, but 8.1% now fall below 70% confidence (up from 0.8%). These marginal predictions are the ones most vulnerable to distribution shift in production.

BERT's confidence collapses entirely. Mean confidence drops from 0.984 to 0.619—barely above random chance for a binary classifier. At FP32, 98.8% of predictions exceed 70% confidence. At INT4, only 3.4% do. The model has gone from "almost always certain" to "always guessing," confirming the mode collapse.

(The companion script ch5/ch5_ptq_failure_diagnostics.py `--confidence` *flag runs this full analysis on your own models.)*

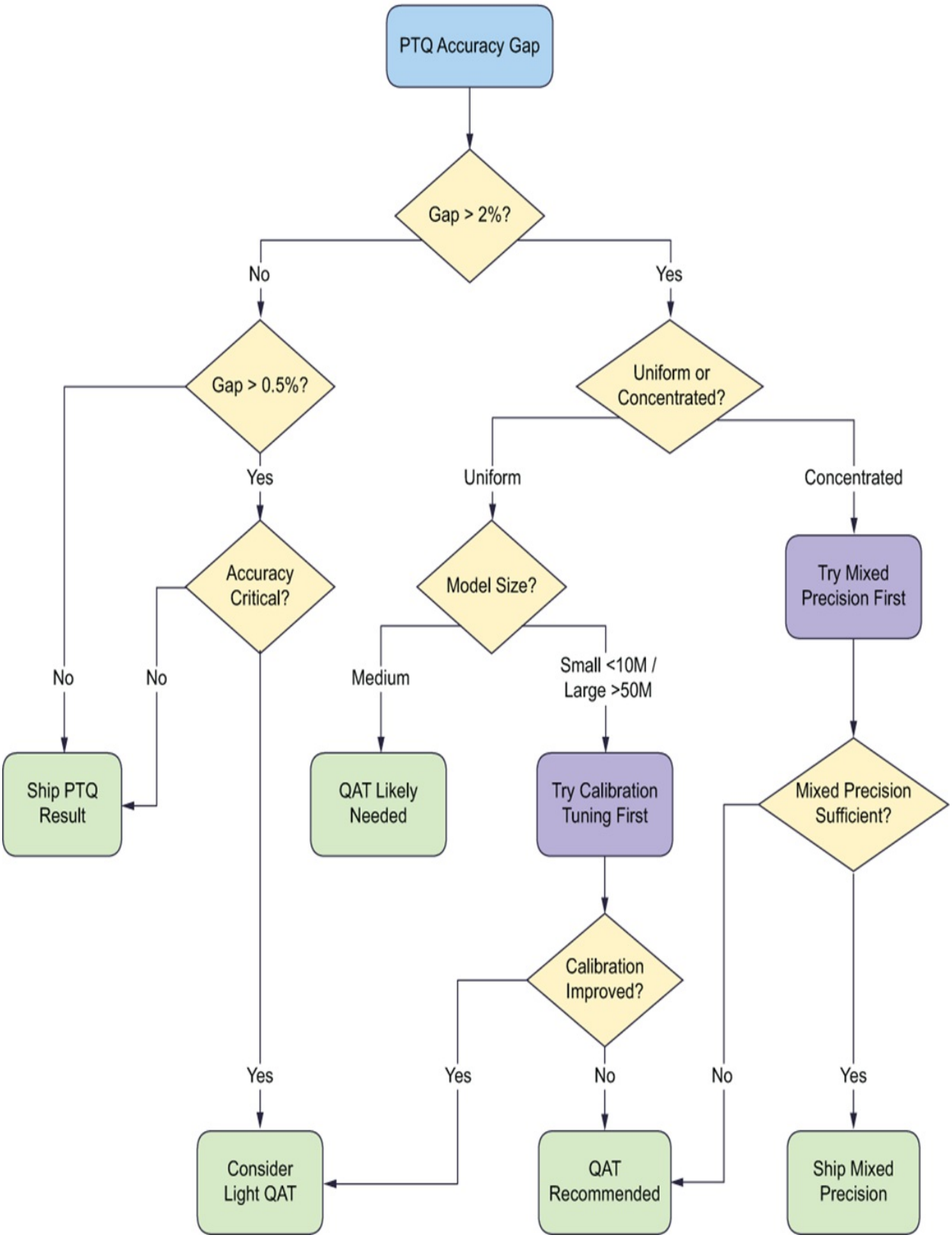
The QAT decision framework

Having diagnosed *how* PTQ fails, we can build a principled decision process. The key insight from our experiments is that the intervention should match the failure mode.

Figure 5.5 encodes this as a decision tree. Start at the top: does your model meet accuracy requirements under PTQ? If yes, stop—deploy PTQ and move on. If not, check the bit-width. INT8 failures almost always trace back to calibration problems (Section 4.3), not fundamental PTQ limitations—revisit your calibration strategy before escalating. For INT4 failures, run the layer sensitivity analysis from Listing 5.2. The critical branch follows: if sensitivity concentrates in a few layers (BERT's pattern), apply mixed precision—keep those layers at INT8 while quantizing the rest to INT4. If sensitivity is diffuse across the network (ResNet-18's pattern), mixed precision has no bottleneck to target, and QAT is the appropriate intervention.

Figure 5.5 QAT decision framework. The key branch is whether failure is concentrated or diffuse—this determines whether mixed precision can save you from the cost of QAT.

PTQ accuracy-gap decision guide



The QAT decision framework encodes several principles from our experiments:

The framework encodes four principles from our experiments.

1. *Always run layer sensitivity before choosing an intervention*—without it, you might apply QAT to a BERT-shaped failure when mixed precision would have sufficed, or try mixed precision on a ResNet-shaped failure that needs whole-model retraining.
2. *INT8 almost never needs QAT*: both models lost under 0.5% at INT8, and an INT8 failure almost always traces back to calibration (Section 4.3) rather than PTQ itself.
3. *INT4 always needs intervention, but the type depends on the architecture*: diffuse failure (ResNet-18's pattern) needs QAT; concentrated failure (BERT's pattern) needs mixed precision.
4. *Order interventions from cheapest to most expensive*: revisit calibration, try mixed precision, then commit to QAT only if the residual gap is still unacceptable. QAT is the right tool when you're targeting INT4 with diffuse sensitivity, when the model is small with little redundancy to absorb error, or when INT8 PTQ is borderline against a tight accuracy budget—and when you have the training infrastructure to run it. If you only have inference access, careful calibration plus mixed precision is your ceiling.

The companion diagnostic script `ch5_ptq_failure_diagnostics.py` *runs all analyses from this section. Use it to diagnose your own models:*

```
# Vision model on ImageNette
python ch5_ptq_failure_diagnostics.py --task vision --model resne

# NLP model on SST-2
python ch5_ptq_failure_diagnostics.py --task nlp --model bert-bas
```

5.2 Train with fake quantization and straight-through estimators

The core challenge is mathematical: quantization involves rounding, and

rounding has zero gradient almost everywhere. A neural network trained with true quantization would receive no learning signal—the gradients would be zero, and the weights would never update. QAT solves this with two complementary techniques: **fake quantization** (which simulates quantization in the forward pass while keeping full precision for gradient computation) and the **straight-through estimator** (which pretends the rounding operation has gradient 1, allowing gradients to flow backward unchanged).

The gradient problem: why quantization breaks backpropagation

Let's start with the fundamental obstacle. The quantization operation we developed in Chapter 2 is:

$$q = round\left(\frac{x}{S}\right) . S$$

where S is the scale factor and q is the quantized-then-dequantized value. The problem is the `round()` function. Its derivative is zero everywhere except at integers, where it's undefined. If we use this true derivative during backpropagation, the gradients vanish.

To make that concrete, listing 5.3 sets up the smallest possible experiment that exposes the failure. We construct a length-4 input tensor with `requires_grad=True`, push it through the quantize-then-rescale operation at a fixed scale of 0.5, sum the result to a scalar loss, and ask autograd what gradient reached `x`. The expected reading: every entry of `x.grad` should be exactly zero, because the only nonzero step in the chain is `torch.round`, and its derivative is zero almost everywhere. A real training loop built on this would receive no learning signal at any step.

Listing 5.3 Demonstrating the zero-gradient problem

```
x = torch.tensor([0.3, 0.7, 1.2, 1.8], requires_grad=True)
scale = 0.5
q = torch.round(x / scale) * scale
loss = q.sum()
loss.backward()
print(f"Gradient: {x.grad.tolist()}")  # All zeros!
```

Output:

```
Input:     [0.30, 0.70, 1.20, 1.80]
Scale:     0.5
Quantized: [0.5, 0.5, 1.0, 2.0]
Gradient:  [0.0, 0.0, 0.0, 0.0]
⚠ All gradients are ZERO - no learning signal!
```

PyTorch's `torch.round()` correctly implements the true gradient. After 100 optimization steps, x hasn't budged - the optimizer received zero gradients at every step.

The straight-through estimator: a useful lie

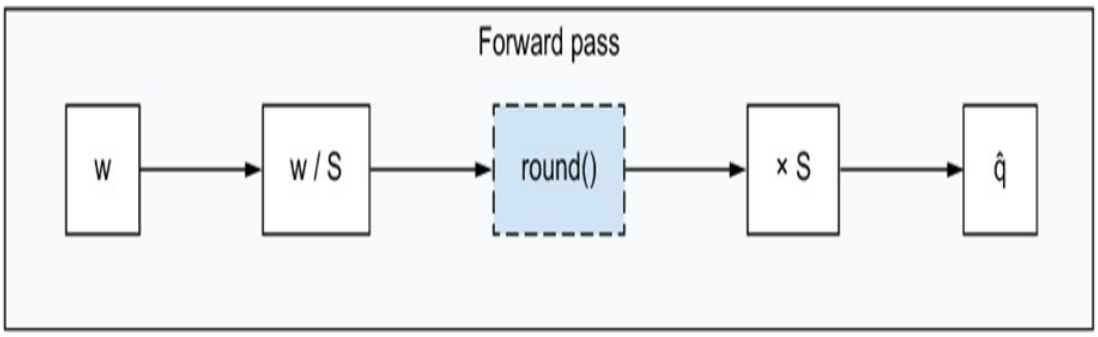
The STE, introduced by[Bengio, Léonard, and Courville (2013)], is the workaround: during the backward pass, pretend the rounding operation didn't happen. Pass gradients through unchanged:

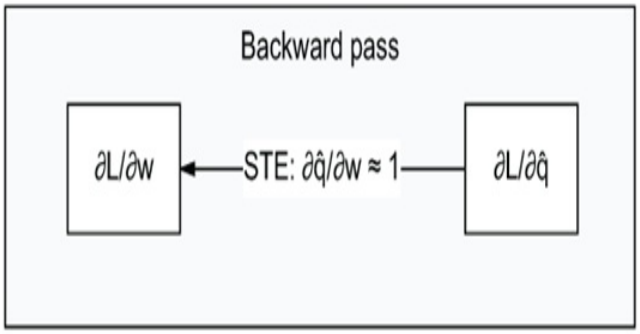
$$\frac{dq}{dx} \approx 1$$

This is "wrong" in the calculus sense - the true gradient is 0. But the direction of the gradient (should this value increase or decrease to reduce loss?) is preserved, even though its exact magnitude through the rounding operation is fabricated.

Figure 5.6 illustrates the mechanism: during the forward pass, values pass through quantization; during the backward pass, gradients skip the rounding entirely and flow straight through.

Figure 5.6 The straight-through estimator passes gradients unchanged through the non-differentiable rounding operation





So, why does this approximation work?

Think of quantization as adding noise to your weights. The true answer through round() is "it doesn't change until you cross an integer boundary" - unhelpful. The STE answer is "the loss changes proportionally to how much you change the weight" - approximately true on average.

Sometimes the STE overestimates the gradient, sometimes it underestimates, but across many weights and steps, the direction is correct. The model learns to position its weights where they'll quantize favorably.

Implementing fake quantization with STE

PyTorch's `torch.autograd.Function` lets us define custom forward/backward behavior.

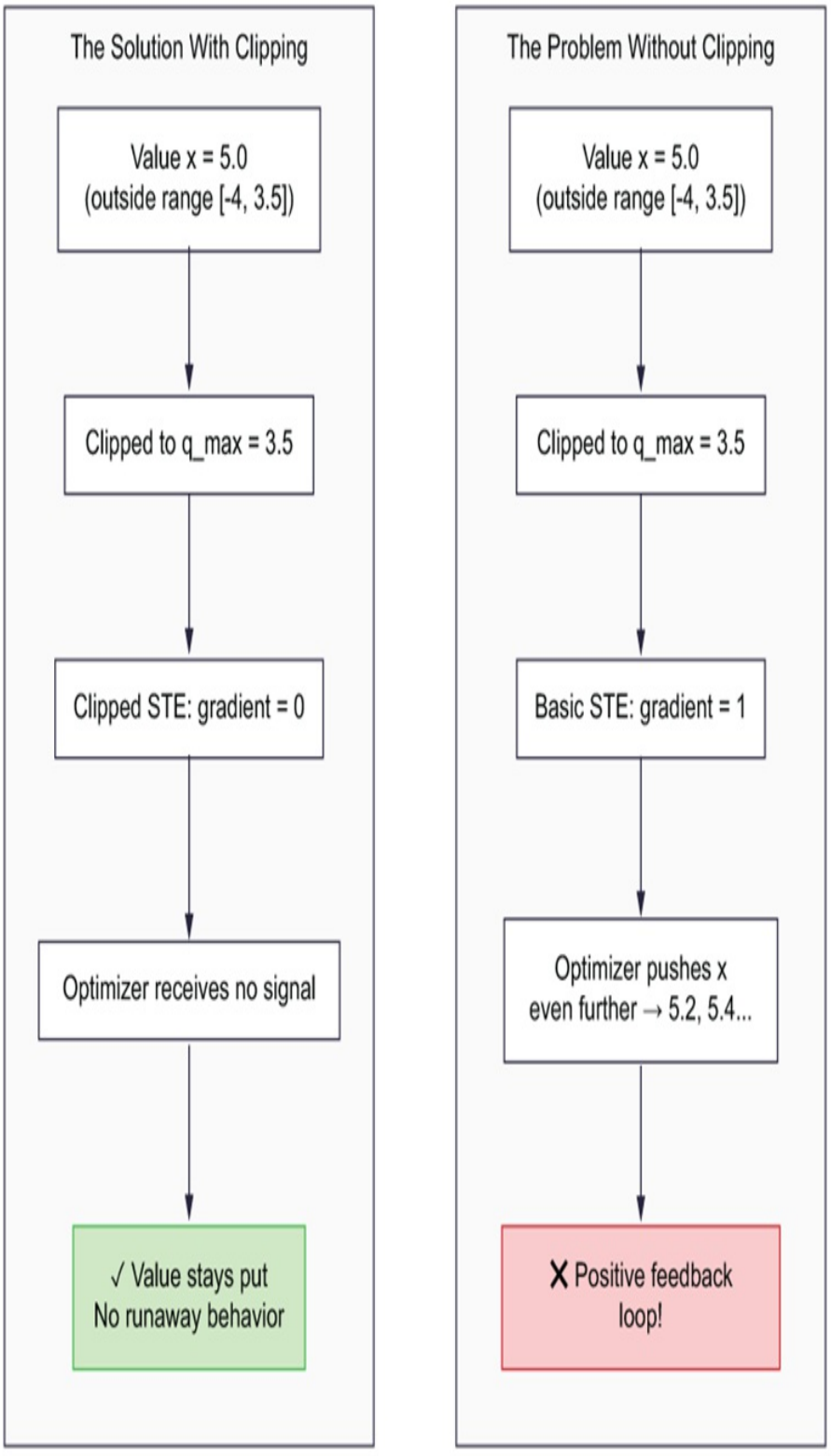
Listing 5.4 implements the full fake quantization operation with the clipped STE variant.

Listing 5.4 Fake quantization with straight-through estimator

```
import torch
import torch.nn as nn

class FakeQuantizeSTE(torch.autograd.Function):
    """Fake quantization with straight-through estimator for grad

    @staticmethod
    def forward(ctx, x, scale, zero_point, q_min, q_max):
        # Quantize: scale to integer range, round, clip
        x_int = torch.round(x / scale + zero_point)
        x_int = torch.clamp(x_int, q_min, q_max)

        # Dequantize: back to floating point
        x_q = (x_int - zero_point) * scale

        # Save for backward (needed for clipping gradient)
        ctx.save_for_backward(x, scale)
        ctx.q_min = q_min
        ctx.q_max = q_max
        ctx.zero_point = zero_point

        return x_q

    @staticmethod
    def backward(ctx, grad_output):
        x, scale = ctx.saved_tensors

        # STE: pass gradient through unchanged
        # But zero out gradients for values that were clipped
        x_int = x / scale + ctx.zero_point
        mask = (x_int >= ctx.q_min) & (x_int <= ctx.q_max)

        grad_input = grad_output * mask.float()

        # No gradient for scale, zero_point, q_min, q_max
        return grad_input, None, None, None, None

def fake_quantize(x, scale, zero_point, q_min, q_max):
    """Convenience wrapper for fake quantization."""
    return FakeQuantizeSTE.apply(x, scale, zero_point, q_min, q_m
```

The backward pass includes the *clipped STE* - we zero out gradients for values that were clipped to q_{min} or q_{max}. Figure 5.7 shows why this matters.

Figure 5.7 Why clipped STE zeros gradients for out-of-range values. A value clipped to the boundary is already saturated—pushing it further cannot change the quantized output. Zeroing the gradient prevents the optimizer from wasting updates on values stuck at the rails, focusing learning capacity on weights that can still move between grid points.



If a value is already outside the representable range and was clipped, pushing it further can't help - the quantized output is already saturated. Without

zeroing the gradient, the optimizer might keep pushing the value further out of range, creating a positive feedback loop.

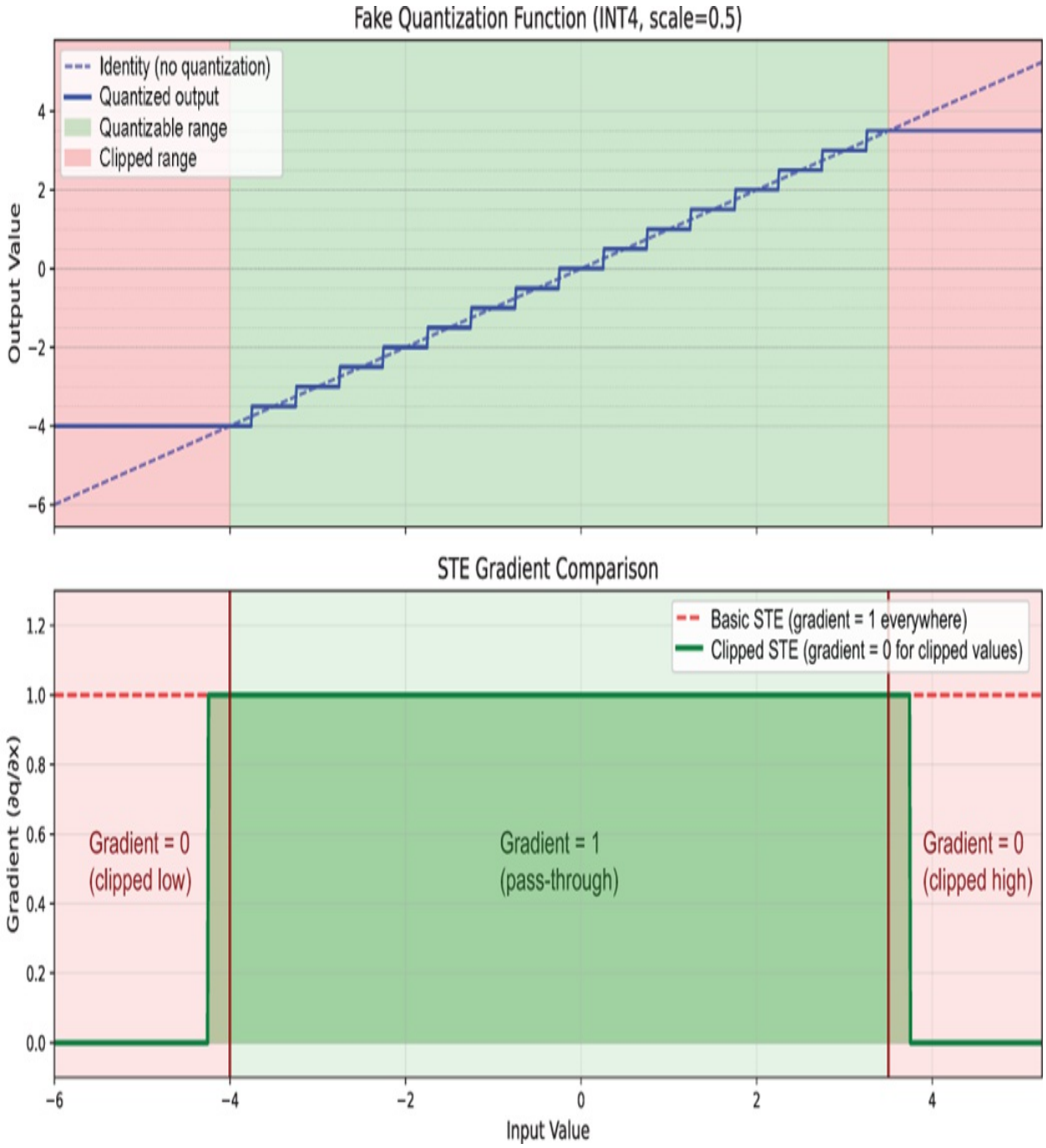
Listing 5.5 verifies that the STE implementation produces the expected gradient pattern—passing gradients for in-range values and zeroing them for clipped ones.

Listing 5.5 Verifying STE gradient flow

```
x = torch.tensor([0.3, 0.7, 1.2, 2.5], requires_grad=True)
scale = torch.tensor(0.5)
zero_point = torch.tensor(0.0)
q_min, q_max = -8, 7  # INT4 symmetric range
q = fake_quantize(x, scale, zero_point, q_min, q_max)
loss = q.sum()
loss.backward()
```

The first three values lie within the representable range [-4.0, 3.5] and receive gradient 1—the STE's "useful lie." The fourth value (2.5) maps to an integer above q_max and gets clipped; the clipped STE zeros its gradient, preventing the optimizer from pushing it further out of range. Figure 5.8 visualizes this boundary behavior across the full input range.

Figure 5.8 Visualizing the clipped STE gradient function. Top: the fake quantization staircase with "quantizable range" and "clipped range." Bottom: gradient behavior - basic STE (dashed) passes gradient 1 everywhere; clipped STE zeros gradients in the clipped regions.



The companion script ch5/ch5_fake_quantization_ste.py runs the full edge-case verification and an optimization experiment that confirms STE actually enables learning: starting from x=0.3 with a target of 1.5, the STE-enabled optimizer reaches the target in 15 steps—the quantized value jumps discretely (0.50 → 1.00 → 1.50) while x moves continuously beneath it. Without STE (true gradients), x doesn't budge.

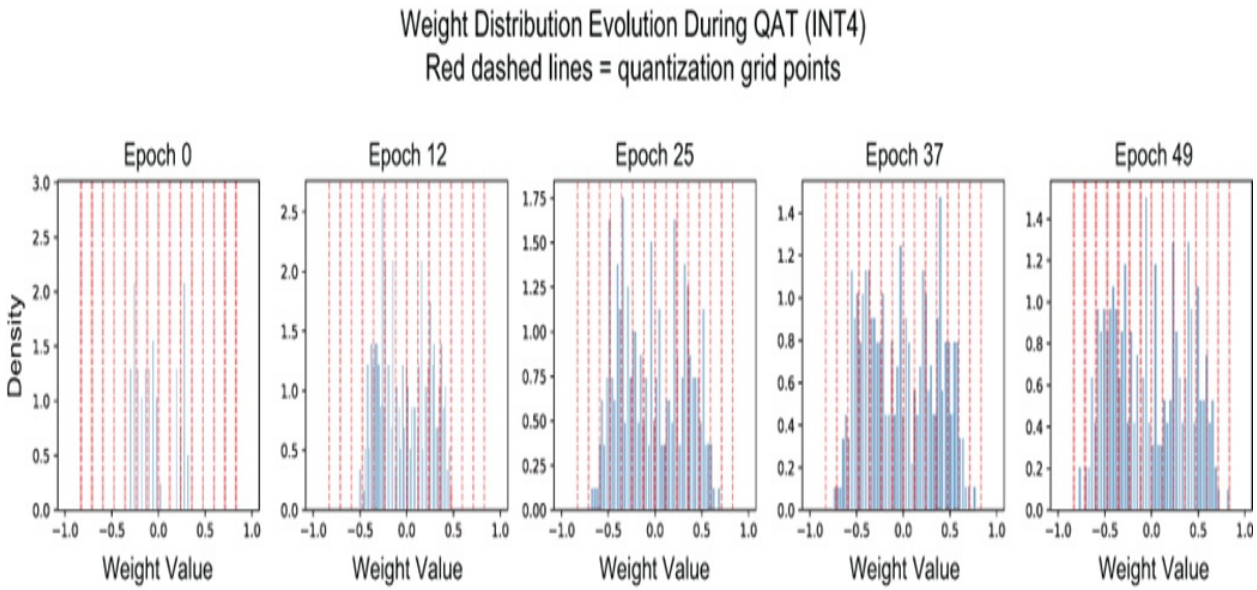
5.2.1 Build a complete fake-quantized layer

With the STE primitive in hand, we build a `FakeQuantizedLinear` module that wraps a standard `nn.Linear`. During the forward pass, it fake-quantizes the weights (per-channel, symmetric—matching Chapter 3's recommendations) and the activations (per-tensor, asymmetric), then computes the linear operation on the quantized values. Crucially, the layer stores weights in full precision: gradients accumulate smoothly in FP32, but the forward pass simulates what deployment will look like. A matching `FakeQuantizedConv2d` does the same for convolutions. *(Both implementations are in ch5_fake_quantize_ste.py.)*

Visualizing what QAT learns

To build intuition, we construct a small CNN—two conv layers, two fully connected layers—where every layer is replaced with its fake-quantized variant. This is the *only* difference from a standard model: `FakeQuantizedConv2d` instead of `nn.Conv2d`, `FakeQuantizedLinear` instead of `nn.Linear`. We train on FashionMNIST at INT4 for 50 epochs and snapshot the weight distributions at regular intervals. Figure 5.9 shows the result—five snapshots of conv1 weights as they reorganize from a smooth Gaussian into sharp spikes aligned with the INT4 grid.

Figure 5.9 Weight distributions migrate toward INT4 grid points during QAT. Five snapshots of conv1 weights (epochs 0–49) show the initially Gaussian distribution reorganizing into sharp spikes aligned with quantization levels (dashed lines), demonstrating that QAT actively adapts weights to minimize rounding error rather than passively tolerating it.



The figure reveals a striking pattern across five snapshots (epochs 0, 12, 25, 37, 49). At epoch 0, the conv1 weights follow a roughly Gaussian distribution. By epoch 25, peaks align with the INT4 quantization grid (dashed lines). By epoch 49, the distribution has transformed into sharp spikes centered on grid points.

This is the core insight of QAT: the model isn't passively tolerating quantization noise; it's actively adapting. Weights migrate toward grid points where they'll incur minimal rounding error during deployment. A weight sitting exactly on a grid point experiences zero quantization error; a weight between grid points gets rounded. Through QAT, the model learns to prefer the former.

The fake quantization observer pattern

In production QAT frameworks (PyTorch's `torch.ao.quantization`, TensorFlow's Model Optimization Toolkit), fake quantization uses an "observer" pattern that separates two concerns: *statistics collection* (what range of values does this tensor span?) and *quantization* (map values to and from that range).

An observer module sits alongside each fake quantizer and tracks running min/max statistics. Two strategies are common: *absolute min/max* tracking

(simple but sensitive to outliers) and *exponential moving average* (smooths over batches, providing more stable scale factors). During training, the fake quantizer queries its observer for the current min/max, computes scale and zero-point using the formulas from Chapter 2 Section 2.2, then applies the quantize-dequantize round trip from Listing 5.4.

This separation matters because it lets you swap observer strategies without touching the quantization logic—a design choice we'll exploit in Section 5.3 when we freeze observers partway through training.

(Full `MinMaxObserver` *and* `FakeQuantize` *module implementations in the companion script* ch5/ch5_ptq_failure_diagnostics.py*.)*

Common STE variants

The basic STE (gradient 1 everywhere) and the clipped STE used in Listing 5.4 (gradient 1 inside the representable range, zero outside) are the two you will encounter in production. Two other variants appear in the literature: the *scaled STE*, which multiplies the backward gradient by the scale factor and is sometimes used at sub-4-bit widths where gradient magnitudes destabilize, and the *tanh STE*, which softens the clipping boundary into a smooth curve. Neither has a measured advantage on the workloads in this book, and every production framework defaults to clipped STE. Use clipped STE unless you have a specific reason not to.

Numerical stability

Three safeguards prevent silent QAT failures. Skip near-zero tensors (their absmax is near zero, making the scale factor explode in the division). Skip near-constant tensors (every value collapses to a single integer, destroying internal structure). Clamp scale factors to a sane range such as [1e-8, 1e6] to keep NaN and Inf out of the graph. All three cost nothing to compute and are implemented in the *StableFakeQuantize* class in the companion script.

5.2.2 The QAT training loop

Putting it all together, Listing 5.6 shows a complete QAT training loop that

incorporates all the machinery we've built—fake quantization insertion, STE gradient flow, and the QAT-specific hyperparameter choices.

Listing 5.6 Complete QAT training loop

```
def prepare_model_for_qat(model, bits=8, inplace=True):
    """Insert fake quantization modules into a model."""
    if not inplace:
        model = copy.deepcopy(model)
    for name, module in model.named_children():
        if isinstance(module, nn.Linear):
            qat_linear = FakeQuantizedLinear(
                module.in_features, module.out_features,
                bits=bits)
            qat_linear.linear.weight.data = \
                module.weight.data.clone()
            if module.bias is not None:
                qat_linear.linear.bias.data = \
                    module.bias.data.clone()
            setattr(model, name, qat_linear)
        else:
            prepare_model_for_qat(module, bits=bits,
                                  inplace=True)
    return model

def qat_training_loop(model, train_loader, val_loader,
                      epochs=10, lr=1e-4, bits=8,
                      device='cuda'):
    """
    QAT training loop. Key differences from standard training:
    1. Lower learning rate (fine-tuning, not training from scratc
    2. Shorter duration (5-20 epochs)
    3. Gradient clipping (STE can amplify magnitudes)
    """
    model = prepare_model_for_qat(model, bits=bits)
    model = model.to(device)
    optimizer = torch.optim.Adam(model.parameters(), lr=lr)
    criterion = nn.CrossEntropyLoss()
    scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(
        optimizer, T_max=epochs)

    best_val_acc = 0.0
    best_model_state = None

    for epoch in range(epochs):
        model.train()
```

```
        for data, target in train_loader:
            data, target = data.to(device), target.to(device)
            optimizer.zero_grad()
            output = model(data)            #A
            loss = criterion(output, target)
            loss.backward()                 #B
            torch.nn.utils.clip_grad_norm_(
                model.parameters(), max_norm=1.0)
            optimizer.step()
        scheduler.step()

        # Validation and best-model tracking
        # ... (standard eval loop, see companion script)

    if best_model_state is not None:
        model.load_state_dict(best_model_state)
    return model
```

QAT-specific practices embedded in this loop: **lower learning rate** (1e-4 to 1e-5, not 1e-3), **shorter duration** (5-20 epochs), **gradient clipping** (STE can produce larger magnitudes, especially at low bit-widths), and **cosine annealing** (smooth decay helps weights settle onto grid points).

The mental anchor for fake quantization and STE:

The STE pretends the gradient through rounding is 1—a "useful lie" whose direction is correct on average. The clipped variant (recommended) additionally zeros gradients for saturated values to prevent runaway optimization. Fake quantization simulates deployment in the forward pass while maintaining full precision for gradient computation—the model learns to position weights on the quantization grid, as Figure 5.9 showed. Numerical stability requires explicit safeguards for near-zero and near-constant tensors. And QAT is fine-tuning, not training from scratch: lower learning rates, shorter durations, and the observer pattern to separate statistics collection from quantization logic.

5.3 Schedule training to minimize cost

QAT isn't free, and the difference between a well-scheduled QAT run and a naïve one can be the difference between a 4-hour fine-tune on a single GPU

and a multi-day burn that still produces worse results.

The cost of QAT is real: every forward pass computes fake quantization on every weight and activation tensor, and every backward pass routes gradients through the STE. But the fake quantization operations are element-wise and memory-bandwidth-bound, making the real cost of QAT about how many epochs you run, not the per-step penalty. Multiply that by unnecessary epochs, and you've turned a practical technique into an expensive one.

We'll address three scheduling decisions: how to warm up the learning rate, when to freeze observer statistics, and how to progressively introduce quantization noise.

5.3.1 Warm up the learning rate

Standard fine-tuning uses a small learning rate, but QAT adds a wrinkle: the model simultaneously adapts to a new objective (task loss computed under fake quantization) and navigates a loss landscape distorted by STE gradient approximation errors. Too large a learning rate causes weights to oscillate around grid points instead of settling onto them—especially at INT4, where the spacing between representable values is wide. Too small a learning rate wastes epochs as weights creep toward grid positions. The solution is a warmup phase followed by cosine decay. Listing 5.7 implements this two-phase schedule.

Listing 5.7 QAT learning rate schedule with warmup

```
def create_qat_lr_schedule(optimizer, num_epochs, warmup_epochs=3
                           min_lr_ratio=0.01):
    """
    QAT-specific LR schedule: linear warmup + cosine decay.

    The warmup phase lets observer statistics stabilize before
    weights begin large updates. The cosine decay gives weights
    time to settle onto grid points.
    """
    def lr_lambda(epoch):
        if epoch < warmup_epochs:
            # Linear warmup: 10% -> 100% of base LR
            return 0.1 + 0.9 * (epoch / warmup_epochs)
```

```
        else:
            # Cosine decay to min_lr_ratio of base LR
            progress = (epoch - warmup_epochs) / (num_epochs - wa
            return min_lr_ratio + (1 - min_lr_ratio) * 0.5 * (
                1 + math.cos(math.pi * progress)
            )

    return torch.optim.lr_scheduler.LambdaLR(optimizer, lr_lambda
```

The warmup serves a QAT-specific purpose: during the first few epochs, fake quantization observers are still calibrating their statistics. Large weight updates against inaccurate quantization parameters push weights into regions that are hard to recover from. The cosine decay is preferable to step decay because the smooth annealing gives weights continuous opportunities for fine adjustments—step decay's sudden drops can cause overshooting onto the wrong grid point.

Freezing observers: the calibration–training handoff

Observer modules (Section 5.2) track running statistics to compute scale factors. During training, these statistics update every batch, which means the quantization grid itself shifts as the model trains. The model adapts its weights to one grid, then the grid moves, then the model adapts again—like trying to park a car while someone keeps moving the parking lines.

The fix is to freeze observers after they've collected enough data. In practice, this means setting each observer to eval mode so it stops updating its running statistics. The quantization grid locks in place; the remaining training epochs let the optimizer nudge FP32 weights so they round to grid points that minimize loss.

When should you freeze? Monitor the relative change in per-layer scale factors across epochs—when the maximum change drops below 1%, the observers have converged. On our SmallCNN, the timing barely matters (all settings from freeze-at-epoch-3 through never-freeze land within 0.2% of each other). But for larger models with wider activation distributions, unfrozen observers introduce a moving target that interacts destructively with the learning rate schedule.

The freeze mechanism costs nothing to implement and provides a safety net for exactly those harder cases. *(Observer freezing and convergence monitoring implementations are in the companion script* ch5/ch5_qat_schedule.py*.)*

Progressive quantization: start gentle, finish precise

The most impactful scheduling decision is whether to introduce quantization noise all at once or progressively. Naïve QAT drops the model straight from FP32 into INT4 - a brutal transition that can spike the loss. Progressive quantization eases the model in, starting at INT8 and stepping down to INT4 partway through training. The intuition: at INT8, most weights already sit close to a grid point. Once comfortable under INT8 constraints, the grid coarsens to INT4 for the final, larger adjustments. Listing 5.8 implements this progressive schedule.The schedule parameters — number of phases, intermediate bit-widths, and epoch allocation — are configurable per model; the $8 \rightarrow 6 \rightarrow 4$ progression shown here is a starting template, not a fixed recipe.

Listing 5.8 Progressive quantization schedule

```
class ProgressiveQuantSchedule:
    """
    Step down bit-width during QAT.
    Example for 20-epoch QAT targeting INT4:
      Epochs  0-4:  INT8 (gentle warm-up)
      Epochs  5-9:  INT6 (intermediate)
      Epochs 10-19: INT4 (target precision)
    """
    def __init__(self, target_bits=4, total_epochs=20):
        self.target_bits = target_bits
        self.total_epochs = total_epochs
        self.schedule = self._build_schedule()

    def _build_schedule(self):
        if self.target_bits >= 8:
            return [(0, 8)]
        warmup_end = self.total_epochs // 4
        mid_end = self.total_epochs // 2
        mid_bits = min(8, self.target_bits + 2)
        return [
            (0, 8),
            (warmup_end, mid_bits),
```

```
            (mid_end, self.target_bits),
        ]

    def get_bits(self, epoch):
        current_bits = self.schedule[0][1]
        for start_epoch, bits in self.schedule:
            if epoch >= start_epoch:
                current_bits = bits
        return current_bits

    def update_model_bits(self, model, epoch):
        """Update fake quantizers to the current bit-width."""
        bits = self.get_bits(epoch)
        for module in model.modules():
            if hasattr(module, 'bits'):
                if module.bits != bits:
                    module.bits = bits
                    # Recompute quantization bounds for new bit-w
                    # ... (update q_min, q_max from bits)
        return bits
```

At inference time, batch norm parameters are typically folded into the preceding conv or linear layer—absorbing the normalization into modified weights and biases. This folding changes the weight distribution: the folded weights have different magnitudes and dynamic range than the unfolded originals. If you run QAT with unfolded batch norm and fold at export time, the quantization parameters you trained against no longer match the actual weights.

The damage is a silent deployment-time mismatch that won't surface during training or FP32 evaluation.

Listing 5.9 implements batch normalization folding—the operation that must happen before fake quantization modules are inserted.

Listing 5.9 Batch normalization folding for QAT

```
def fold_bn_into_conv(conv, bn):
    """Fold batch norm into preceding Conv2d."""
    w = conv.weight.data
    b = conv.bias.data if conv.bias is not None else torch.zeros(
        w.shape[0], device=w.device)
    factor = bn.weight / torch.sqrt(bn.running_var + bn.eps)  #A
```

```
    new_weight = w * factor.view(-1, 1, 1, 1)
    new_bias = (b - bn.running_mean) * factor + bn.bias
    new_conv = nn.Conv2d(conv.in_channels, conv.out_channels,
                         conv.kernel_size, conv.stride,
                         conv.padding, bias=True)
    new_conv.weight.data = new_weight
    new_conv.bias.data = new_bias
    return new_conv
```

5.3.2 Put the schedule together

A complete QAT schedule chains these decisions into three phases:

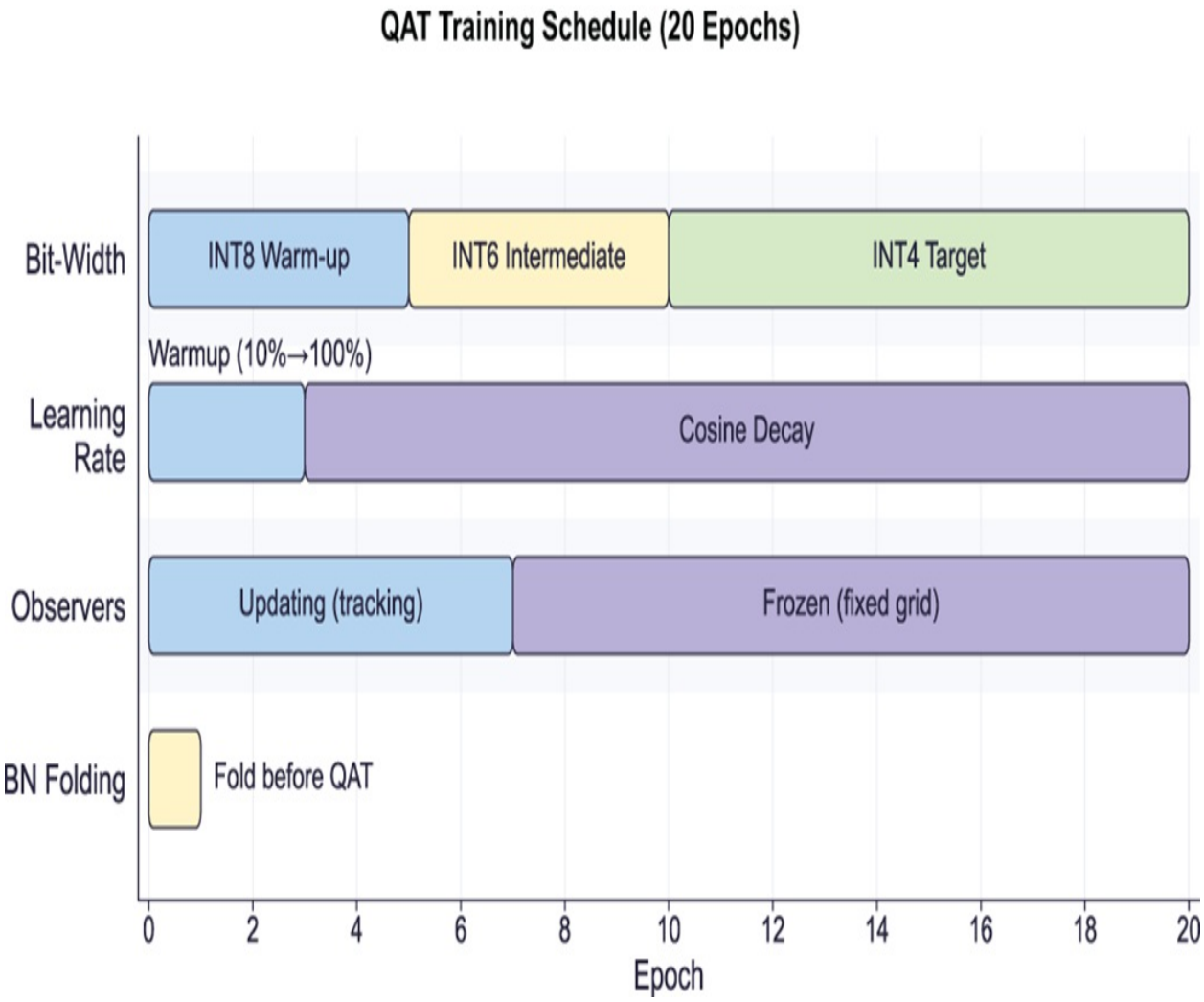
Phase 1: Calibration (epochs 1–5): Batch normalization is folded before training begins. INT8 with warming-up learning rate. Observers actively tracking statistics.

Phase 2: Transition (epochs 6–10): Bit-width drops to INT6. Observers freeze at epoch 8, locking in the quantization grid.

Phase 3: Target precision (epochs 11–20): Bit-width drops to INT4. Loss spikes at the transition then recovers. Cosine decay ensures weights settle precisely onto INT4 grid points.

Figure 5.10 visualizes how these four components—bit-width, learning rate, observer state, and loss—interleave across the 20-epoch schedule.

Figure 5.10 The complete QAT schedule coordinates four components. Bit-width decreases progressively (INT8→INT6→INT4). The learning rate warms up during INT8 calibration, then cosine-decays through the remaining phases. Observers freeze at epoch 8, after they've seen the INT6 distribution. Batch normalization is folded before training begins.



Listing 5.10 chains these decisions into a single function you can call on any model. Reading top to bottom, it folds batch normalization, prepares the model for QAT at INT8, builds the warmup-plus-cosine LR schedule, instantiates the progressive bit-width schedule, and enters the epoch loop. Three lines inside the loop carry the entire scheduling logic: `update_model_bits` advances bit-width at the scheduled epochs, `apply_observer_freeze_schedule` flips observers to eval mode at `freeze_epoch`, and `scheduler.step()` advances the learning rate. Watch how the components interleave rather than how each one works in isolation.

Listing 5.10 Complete scheduled QAT pipeline

```
def scheduled_qat_pipeline(model, train_loader, val_loader,
                           device, target_bits=4, total_epochs=20
```

```
                          base_lr=1e-4, warmup_epochs=3,
                          freeze_epoch=7):
    model = fold_all_bn(model)
    model = prepare_model_for_qat(model, bits=8)
    optimizer = torch.optim.Adam(model.parameters(), lr=base_lr)
    scheduler = create_qat_lr_schedule(
        optimizer, total_epochs, warmup_epochs)
    prog_schedule = ProgressiveQuantSchedule(
        target_bits=target_bits, total_epochs=total_epochs)

    for epoch in range(total_epochs):
        current_bits = prog_schedule.update_model_bits(
            model, epoch)
        apply_observer_freeze_schedule(
            model, epoch, freeze_epoch)
        # ... training step with gradient clipping
        scheduler.step()
    return model
```

Running the complete pipeline on SmallCNN, the key epochs from the training log:

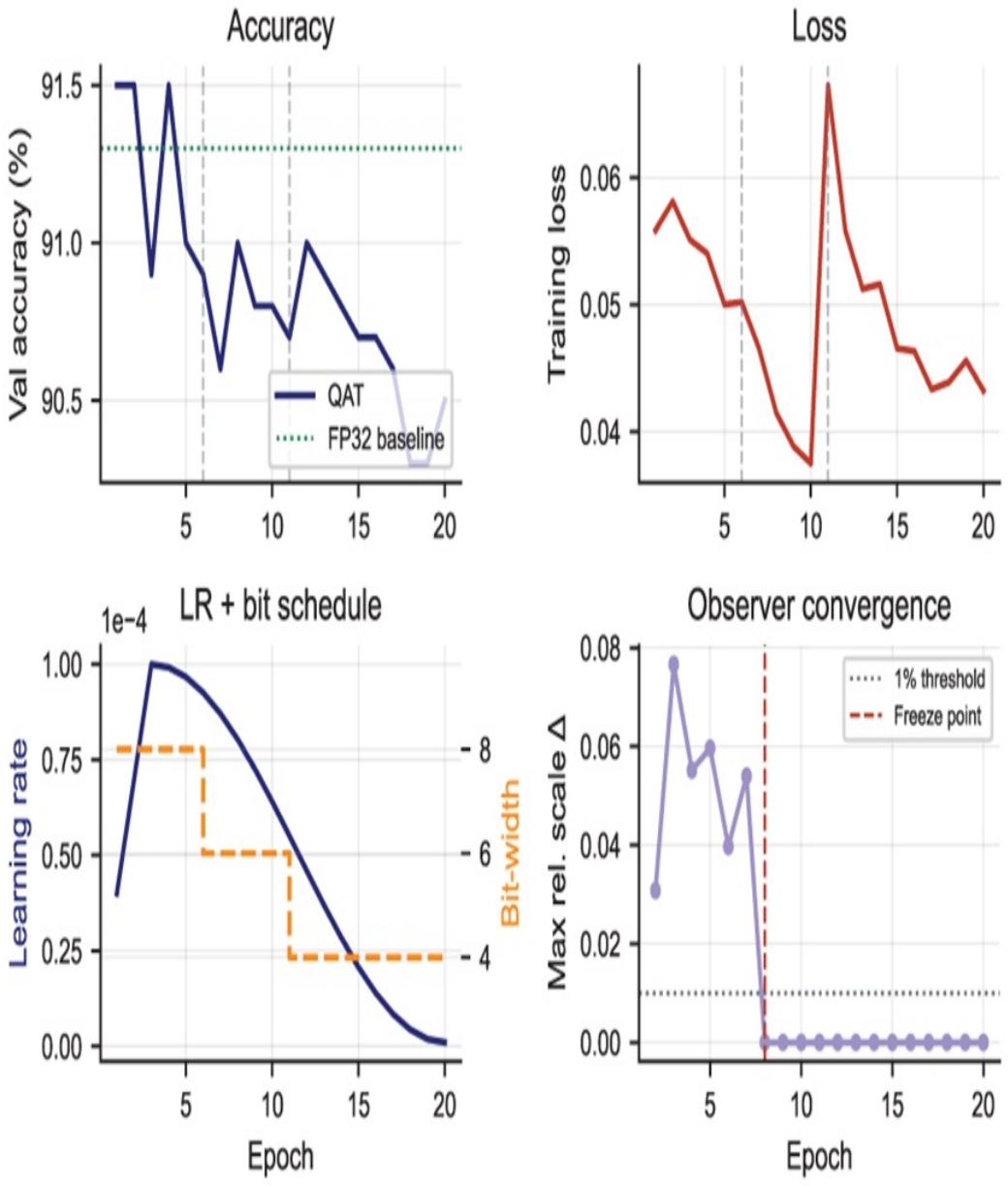
```
Scheduled QAT Pipeline (SmallCNN → INT4, 20 epochs):
FP32 baseline: 90.00%
```

Epoch	Bits	LR	Loss	Val Acc	Obs	ΔScale
1	INT8	0.000040	0.0751	89.50%	active	inf
3	INT8	0.000100	0.0697	89.50%	active	0.0218
6	INT6	0.000093	0.0646	90.00%	active	0.0096
8	INT6	0.000080	0.0549	89.40%	frozen	0.0000
11	INT4	0.000055	0.0844	89.30%	frozen	0.0000
14	INT4	0.000028	0.0605	90.30%	frozen	0.0000
20	INT4	0.000001	0.0516	89.70%	frozen	0.0000

```
Scheduled QAT: 90.30% (best)  |  Accuracy gap: +0.30% vs FP32
```

The INT6 transition at epoch 6 barely causes a dip. The INT4 transition at epoch 11 is harder: loss spikes from 0.0498 to 0.0844 and accuracy dips to 89.30%. But by epoch 14 it peaks at 90.30% - slightly above the FP32 baseline. QAT acts as a form of regularization (the fake quantization noise prevents overfitting), so slightly exceeding FP32 isn't unusual. Figure 5.11 shows the four-panel view of the full training run—accuracy, loss, bit-width transitions, and learning rate decay across all 20 epochs.

Figure 5.11 The four-panel view of the scheduled QAT pipeline. Top left: validation accuracy oscillates but stays within ~1 point of FP32 despite dropping to INT4. Top right: training loss spikes at each bit-width transition (dashed verticals at epochs 6 and 11), then recovers. Bottom left: the learning rate (solid) warms up and cosine-decays while bit-width (dashed steps) progresses 8→6→4. Bottom right: observer ΔScale spikes at bit-width transitions, then drops to zero after freezing at epoch 8.



Cost analysis, what you spend and what you save:

```
Cost Comparison (SmallCNN on FashionMNIST, 5k subset):
═══════════════════════════════════════════════════════════════════
Method                | Final Acc  | Best Acc  |  →85%  |  Time
```

FP32 Baseline	90.50%	90.50%	—	—
PTQ (no training)	88.60%	88.60%	—	~0 s
Direct INT4 QAT	90.80%	90.80%	1ep	30.1 s
Scheduled QAT	91.00%	91.00%	1ep	30.1 s

PTQ at INT4 loses 1.9 points from FP32. Both QAT approaches recover the gap fully. Scheduled QAT edges direct QAT by 0.20% - a small margin within run-to-run noise for this model. Both take identical wall-clock time (~30 seconds for 20 epochs on CPU).

On a small CNN with FashionMNIST, the individual scheduling components have modest impact—both QAT approaches recover the PTQ gap fully, and scheduled QAT edges direct QAT by just 0.20%. This is typical of small-model experiments. The reason to learn these techniques: they are *free* (zero additional compute), the gains compound at scale, and the scheduled approach produces a more stable, debuggable trajectory. Direct QAT's observer ΔScale never settles, oscillating 1–5% through all 20 epochs; scheduled QAT freezes observers and converges predictably.

Practical guidelines

Fold batch normalization before QAT—this is a correctness requirement, not an optimization. Warm up the learning rate for 10–15% of total epochs, starting at 10% of the base LR. Freeze observers once the max relative scale change drops below 1%; in progressive QAT, freeze a few epochs after the penultimate bit-width transition. Use progressive quantization for INT4 and below, allocating roughly 25% of epochs to INT8 warm-up, 25% to an intermediate bit-width, and 50% to the target. Budget 15–25 epochs total—if accuracy hasn't converged by then, the problem is likely the quantization scheme or architecture, not training duration. Use cosine annealing with minimum LR at ~1% of base.

The companion script ch5/ch5_qat_schedule.py *implements the complete scheduled pipeline and all experiments from this section:*

```
# Run full scheduled QAT pipeline
python ch5_qat_schedule.py --mode full --target-bits 4 --epochs 2
```

```
# Compare scheduling strategies
python ch5_qat_schedule.py --mode compare --target-bits 4

# Demonstrate observer freezing impact
python ch5_qat_schedule.py --mode observer-freeze --target-bits 4

# Demonstrate BN folding order
python ch5_qat_schedule.py --mode bn-folding

# Generate all figures for this section
python ch5_qat_schedule.py --mode full --target-bits 4 --save-plo
```

5.4 Use per-channel strategies for convolution and linear layers

Chapter 3 established that per-channel quantization eliminates the outlier penalty—each output channel gets its own scale factor, recovering 3–9× better MSE than per-tensor at negligible overhead. That analysis measured *static* quantization error on frozen weights. Now we need to answer a different question: does the granularity of your fake quantizer affect how well QAT *trains*?

The mechanism is the STE. Recall from Section 5.2 that the straight-through estimator passes gradients only where values aren't clipped to the quantization boundaries. Per-tensor fake quantization uses a single scale derived from the global weight maximum. Channels with small magnitudes are squeezed into a tiny fraction of the integer range—and their weights pile up at the clipping boundaries, where the STE kills gradients entirely. Per-channel quantization gives each channel its own scale, which means each channel uses the full integer range and fewer weights get clipped. The result: more gradients survive, and every channel can adapt independently during training.

5.4.1 The gradient survival problem

Consider ResNet-18's first convolutional layer with shape `[64, 3, 7, 7]`—64 output channels, each a 3×7×7 filter. Under INT4 per-tensor quantization, a single scale factor is derived from the global maximum across all 64

channels. Channels whose weights are much smaller than the maximum use only a handful of the 15 available quantization levels. Weights at the boundaries get clipped, and the STE zeros out their gradients. Listing 5.11 implements per-channel fake quantization with the STE, broadcasting a separate scale per output channel.

Listing 5.11 Per-channel fake quantization for QAT

```
class FakeQuantizePerChannel(torch.autograd.Function):
    """Per-channel fake quantization with STE."""
    @staticmethod
    def forward(ctx, weight, scales, q_min, q_max, axis):
        shape = [1] * weight.dim()
        shape[axis] = -1                                    #A
        scales_bc = scales.view(shape)

        w_q = torch.clamp(torch.round(weight / scales_bc),
                          q_min, q_max)
        w_dq = w_q * scales_bc
        mask = (w_q > q_min) & (w_q < q_max)               #B
        ctx.save_for_backward(mask)
        return w_dq

    @staticmethod
    def backward(ctx, grad_output):
        mask, = ctx.saved_tensors
        return grad_output * mask.float(), None, None, None, None
```

The key difference from the per-tensor version in Listing 5.4 is the broadcasting. Instead of a scalar scale, we have a vector of scales—one per channel—that reshapes to broadcast across all the spatial and input-channel dimensions. The STE mask is computed element-wise, so each channel's clipping boundary is determined by its own scale.

The scale computation follows the same per-channel logic from Chapter 3: flatten each channel to a vector, take its absolute max, divide by q_max. One QAT-specific detail is critical:

Listing 5.12 Per-channel scale computation for conv and linear layers

```
def compute_per_channel_scales(weight, bits=8, axis=0):
    q_max = (1 << (bits - 1)) - 1
```

```
    flat = weight.detach().reshape(weight.shape[0], -1)  #A
    abs_max = flat.abs().amax(dim=1)                     #B
    scales = abs_max / q_max
    return torch.where(scales == 0, torch.ones_like(scales), scal
```

The scales are treated as constants for each forward pass, recomputed fresh at every step as weights evolve. We don't want gradients flowing through the scale computation—only through the fake quantization itself.

The fake-quantized linear layer from Listing 5.8 used per-tensor quantization. Upgrading to per-channel requires exactly two changes: compute per-channel scales, and call the per-channel fake quantizer:

Listing 5.13 Fake-quantized conv layer with configurable granularity

```
class FQConv2d(nn.Module):
    """Conv2d with per-tensor or per-channel fake quantization."""
    def __init__(self, in_ch, out_ch, kernel_size,
                 padding=0, bits=8, per_channel=False):
        super().__init__()
        self.conv = nn.Conv2d(in_ch, out_ch, kernel_size,
                              padding=padding)
        self.bits = bits
        self.per_channel = per_channel
        self.q_max = (1 << (bits - 1)) - 1
        self.q_min = -self.q_max

    def forward(self, x):
        w = self.conv.weight
        if self.per_channel:
            scales = compute_per_channel_scales(w, self.bits)
            w_fq = FakeQuantizePerChannel.apply(
                w, scales, self.q_min, self.q_max, 0)          #A
        else:
            scale = w.detach().abs().max() / self.q_max
            w_fq = FakeQuantizePerTensor.apply(
                w, scale, self.q_min, self.q_max)
        return nn.functional.conv2d(
            x, w_fq, self.conv.bias,
            self.conv.stride, self.conv.padding)
```

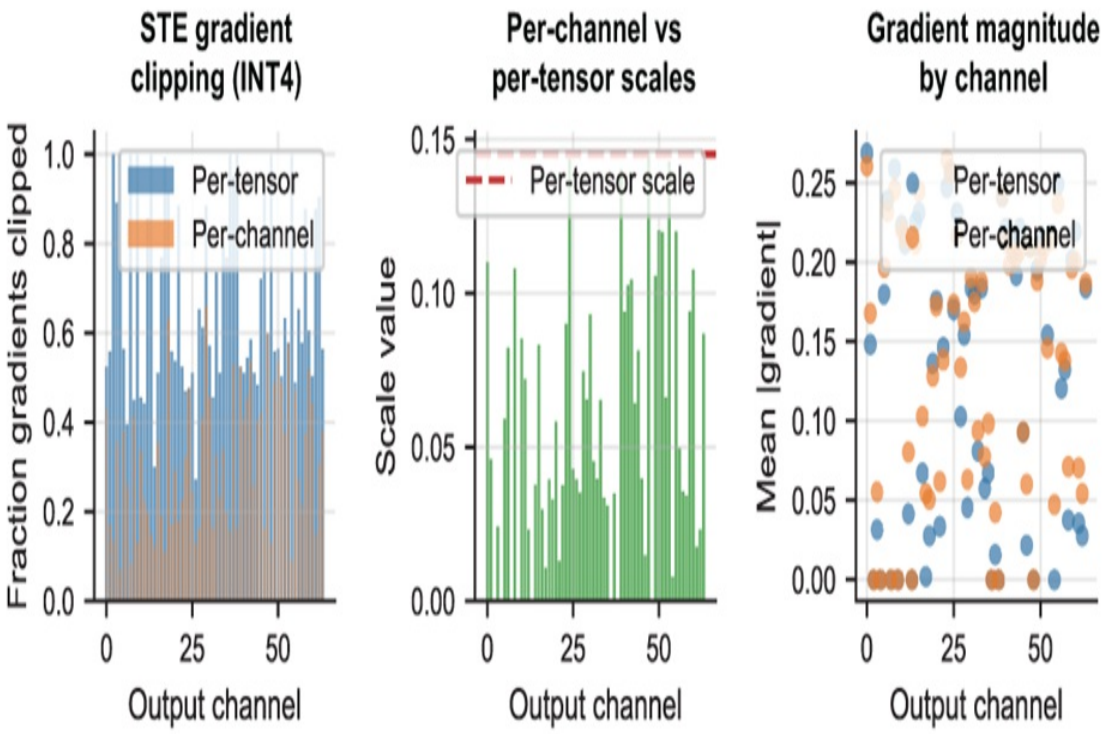
The `FQLinear` variant is identical except it wraps `nn.Linear` and calls `nn.functional.linear`. Both share the same scale computation and fake quantizer.

The following experiment applies INT4 fake quantization to ResNet-18's `conv1` weights (pretrained, shape `[64, 3, 7, 7]`) and compares how many gradients survive the STE under per-tensor vs per-channel:

```
INT4 fake quantization gradient analysis:
------------------------------------------------------
Per-tensor:  avg 67.4% gradients clipped per channel
Per-channel: avg 31.5% gradients clipped per channel
Gradient survival improvement: 2.10×

Worst 5 channels (% gradients clipped):
   Channel    Per-tensor   Per-channel
        48       100.0%        12.9%
         2       100.0%        13.6%
         4       100.0%         6.8%
        36       100.0%        15.6%
         7       100.0%         8.2%
```

Under per-tensor quantization, 67.4% of gradients are zeroed out on average —the STE kills two-thirds of the learning signal. Five channels lose *all* their gradients (100% clipped). These channels are effectively frozen; they cannot adapt during QAT. Per-channel quantization cuts the clipping rate roughly in half (31.5%) and—critically—no channel is fully silenced. Even the worst per-channel case (channel 36 at 15.6%) retains over 84% of its gradients. Figure 5.12 visualizes this gap across all 64 channels, alongside the per-channel scale ratios that explain why certain channels are disproportionately affected under per-tensor quantization.

Figure 5.12 Three views of the gradient survival problem on ResNet-18 conv1 at INT4. Left: fraction of gradients clipped per output channel—per-tensor clips 100% of gradients in several channels, while per-channel stays well below 20%. Center: per-channel scales vs the single per-tensor scale —many channels have scales an order of magnitude smaller than the global maximum. Right: per-channel quantization produces more uniform gradient magnitudes across channels, while per-tensor leaves some channels with zero gradient signal.



Why does the scale ratio reach 80 million×? Because some channels in ResNet-18's first layer have near-zero weight magnitudes—their absolute maximum is ~10^{-8} while the global maximum is 0.145. Per-tensor quantization maps all these tiny weights to integer 0. Per-channel quantization gives each such channel its own microscopic scale, preserving its internal structure.

5.4.2 Measure the training impact

The gradient analysis above uses pretrained ResNet-18 weights and shows a dramatic 2.1× improvement in gradient survival. But the real question is whether this translates to accuracy gains during QAT. We train our SmallCNN from Section 5.2 under three conditions: FP32 (no quantization), INT4 per-tensor QAT, and INT4 per-channel QAT—identical hyperparameters (Adam, lr=1e-3, 15 epochs, 5k FashionMNIST subset, same random seed):

Listing 5.14 Comparing per-tensor vs per-channel QAT accuracy

```
results = {}
```

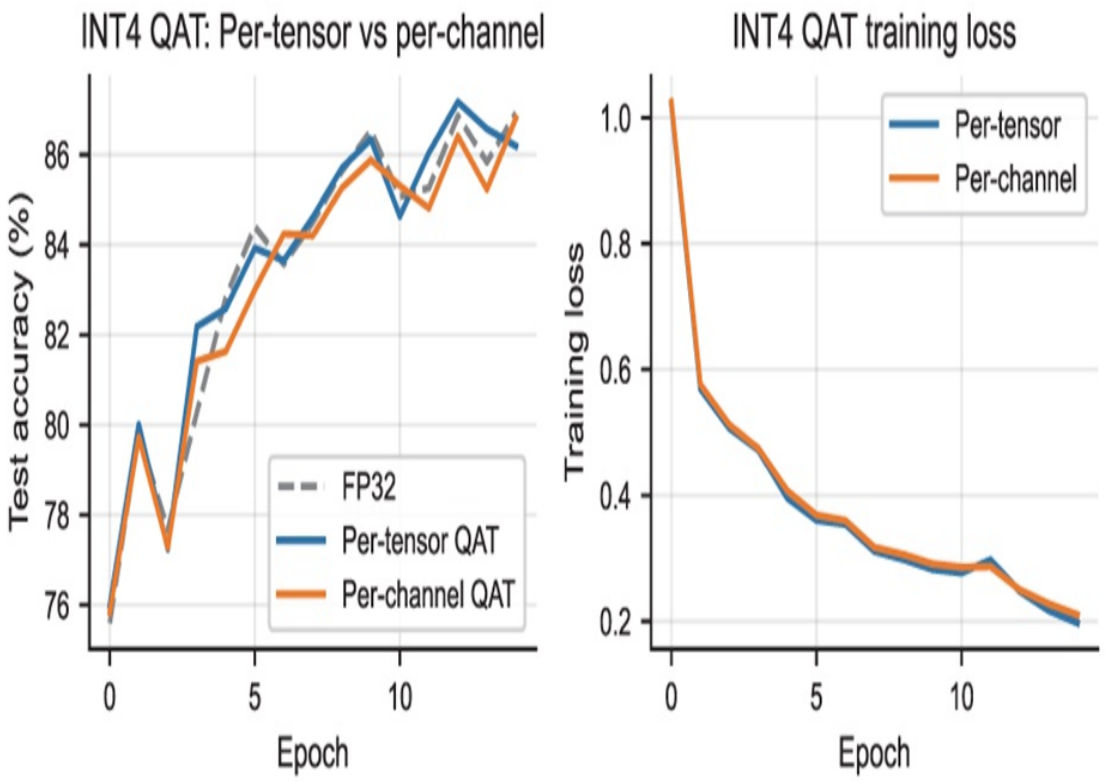
```
for name, per_channel in [('per-tensor', False),
                          ('per-channel', True)]:
    torch.manual_seed(42)
    model = SmallCNN(bits=4, per_channel=per_channel)           #
    optimizer = optim.Adam(model.parameters(), lr=1e-3)

    for epoch in range(15):
        model.train()
        for batch_x, batch_y in train_loader:
            optimizer.zero_grad()
            loss = criterion(model(batch_x), batch_y)
            loss.backward()                                     #
            optimizer.step()

    results[name] = evaluate(model, test_loader)
```

Running this experiment:

```
  Method               Best Acc   Final Acc   Gap vs FP32
──────────────────────────────────────────────────────────
  FP32                  86.92%     86.92%       +0.00%
  per-tensor            87.17%     86.19%       +0.25%
  per-channel           86.81%     86.81%       -0.11%
══════════════════════════════════════════════════════════

  Per-channel advantage: -0.36%
```

The numbers tell a clear story: all three methods converge to within 1 point of each other, and per-channel shows no advantage on this model. Figure 5.13 plots the full training curves to confirm—accuracy and loss trajectories are virtually indistinguishable across all three conditions.

Figure 5.13 Left: Test accuracy curves for FP32, per-tensor INT4 QAT, and per-channel INT4 QAT. All three converge to nearly the same accuracy (~87%). Right: Training loss curves are virtually indistinguishable between the two QAT strategies.



On our SmallCNN with FashionMNIST, per-tensor and per-channel QAT produce statistically indistinguishable results—both reach ~87%, both within noise of the FP32 baseline. Per-tensor even edges per-channel by 0.36% on best accuracy, well within run-to-run variance.

This might seem to contradict the dramatic 2.1× gradient survival improvement we measured on ResNet-18 `conv1`. It doesn't—it illustrates an important principle. The gradient survival gap matters most when channels have *widely varying* weight magnitudes. ResNet-18's pretrained `conv1` exhibits exactly this: an 80-million× ratio between the largest and smallest per-channel scales. Our randomly initialized SmallCNN doesn't have this problem. Its conv1 channels have a tight scale range of [0.033, 0.048]—barely 1.4× between the largest and smallest. When all channels are similarly sized, per-tensor and per-channel produce similar STE masks and similar training dynamics.

This matches a pattern we've seen throughout the book: quantization techniques that are essential for large, pretrained models can be overkill for small, randomly initialized ones. Per-channel QAT matters most for fine-

tuning pretrained models where weight distributions have diverged across channels, for large models (7B+ parameters) where some channels develop extreme magnitudes (the "outlier channels" from Chapter 3, Section 3.3), and at very low bit-widths where even modest channel variation exhausts the limited quantization levels.

NOTE

The reason to learn per-channel QAT despite the small-model result: it's *free* (zero additional compute), all production runtimes support it, and it becomes essential at scale. Per-channel is the safe default—per-tensor is only warranted when your target hardware explicitly doesn't support per-channel operations.

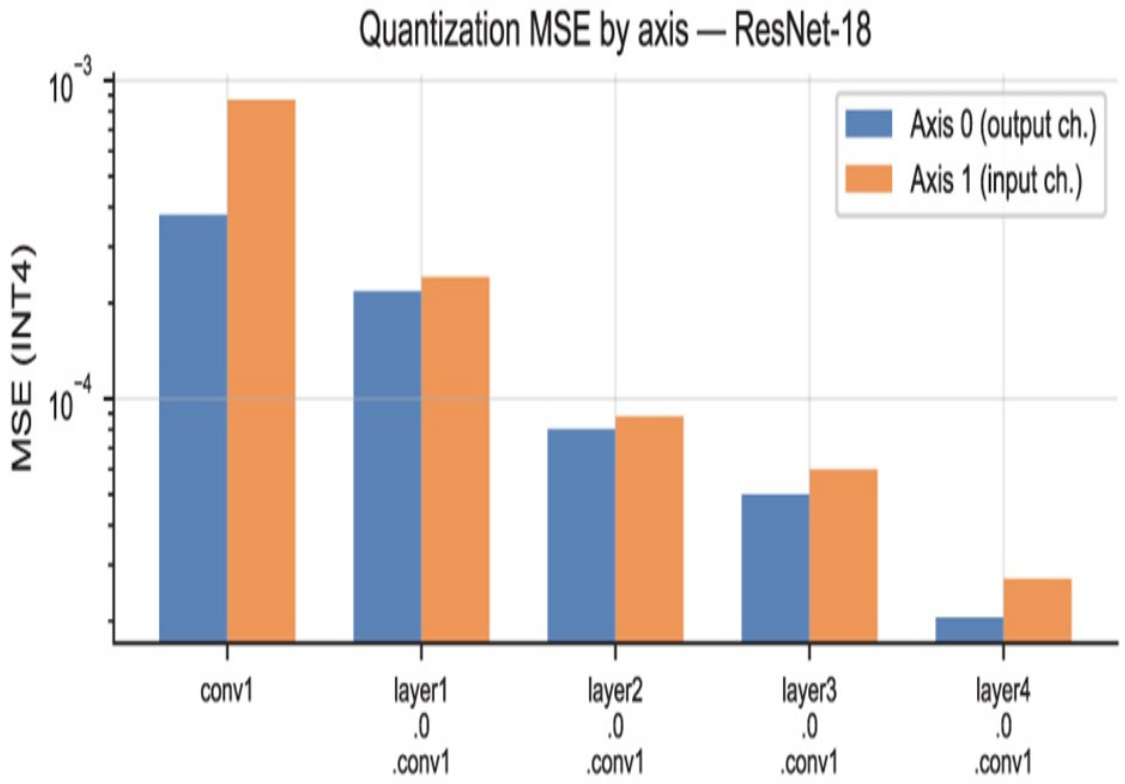
Per-channel scale stability during training

A reasonable concern with recomputing scales every forward pass: do they drift enough to destabilize training? On SmallCNN, no—the companion script tracks all 32 conv1 scales across 15 epochs and they don't move. Abs-max calibration captures the single most extreme weight per channel, and QAT's gradient updates redistribute mass toward grid points (Figure 5.9) without touching the extremes. Pretrained models under QAT fine-tuning can show modest drift in early epochs, then stabilization; the observer freezing schedule from Section 5.3 is the safeguard.

The convolution axis question: output channels, always

Conv2d weights have shape [out_channels, in_channels, kernel_h, kernel_w]. Per-channel quantization along axis 0 (output channels) is standard, and the companion script *ch5/ch5_per_channel_qat.py* confirms it wins in every ResNet-18 layer. The advantage is most pronounced in conv1 (2.3× lower MSE than axis 1), where the asymmetry between 64 output channels and 3 input channels is extreme—quantizing along axis 1 gives just 3 scales for 64 different feature detectors. In deeper layers where both dimensions are large, the advantage narrows to 1.1–1.3×, but axis 0 still consistently wins. For depthwise convolutions (shape [channels, 1, kH, kW]), both axes are equivalent. Figure 5.14 visualizes the comparison.

Figure 5.14 INT4 quantization MSE by axis for five ResNet-18 convolutional layers. Axis 0 produces lower MSE than axis 1 in every layer. The advantage is most pronounced in conv1, where the asymmetry between 64 output and 3 input channels is extreme.



The special case is depthwise convolution, where the weight shape is `[channels, 1, kH, kW]`—each output channel has exactly one input channel. Here axis 0 and axis 1 are equivalent (MSE: 7.54e-05 for a 128-channel depthwise layer, same for both axes). MobileNet and EfficientNet practitioners don't need to worry about this choice for their depthwise layers.

Practical guidelines

Always use per-channel quantization for weights regardless of bit-width—it costs nothing in training, has negligible-to-essential accuracy benefit depending on model scale, and every production runtime supports it (TensorRT, ONNX Runtime, TFLite). There is no reason to use per-tensor for weights. Keep activations per-tensor, or per-token for transformers (Chapter 3, Section 3.3)—activation channels are input-dependent and would require dynamic per-channel scales that change every batch.

Quantize along axis 0 for both conv and linear layers (output channels and output features, respectively). And expect scale stability during QAT: abs-max per-channel scales tend to be rock-solid, so for random initialization, observers may never need freezing.

The companion script ch5/ch5_per_channel_qat.py implements all experiments from this section:

```
# Gradient flow analysis (per-tensor vs per-channel STE)
python ch5_per_channel_qat.py --mode gradient-analysis

# Per-channel scale evolution during training
python ch5_per_channel_qat.py --mode scale-evolution

# Accuracy comparison: per-tensor vs per-channel QAT
python ch5_per_channel_qat.py --mode accuracy-compare

# Convolution axis analysis on ResNet-18
python ch5_per_channel_qat.py --mode conv-axis

# Run all experiments and generate figures
python ch5_per_channel_qat.py --mode all --save-plots
```

5.5 Adapt transformers efficiently

Sections 5.2–5.4 built the QAT toolkit on convolutional networks. Transformers share the same building blocks—linear projections, nonlinearities, residual connections—but arrange them differently. Each encoder block has six linear layers: query, key, value, and output projections in attention, plus an up-projection and down-projection in the FFN. Not all six respond equally to fake quantization. The engineering question is: which get fake quantizers, and which stay in FP32?

Run a quick sublayer diagnostic

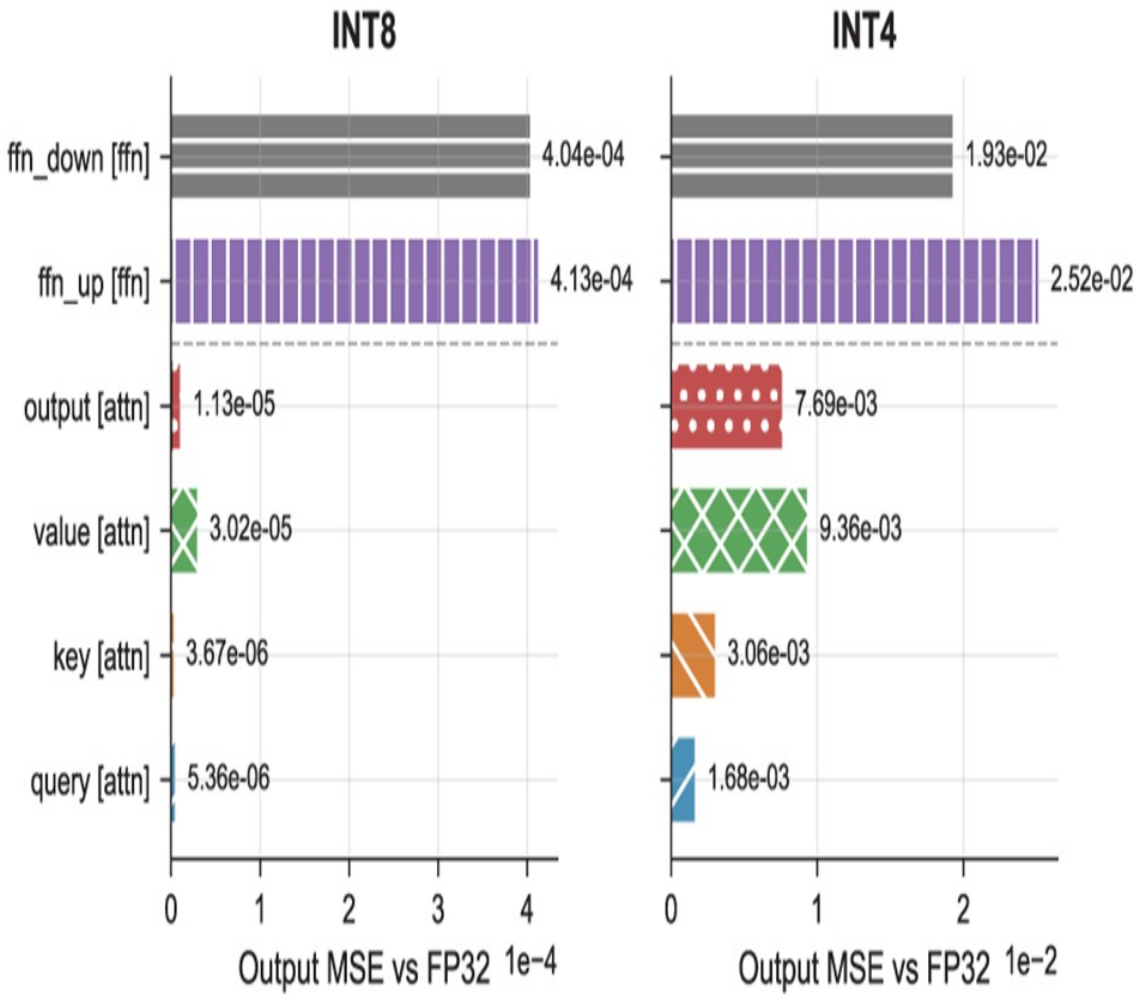
Before committing to a QAT training run, profile which sublayer types hurt most. The method is the same perturbation analysis from Section 5.1 (Listing 5.2): fake-quantize one sublayer type at a time across all 12 encoder blocks, measure output MSE against the FP32 baseline.

Listing 5.15 Transformer sublayer sensitivity profiling

```
sublayer_paths = OrderedDict([
    ("query",    "attention.self.query"),
    ("key",      "attention.self.key"),
    ("value",    "attention.self.value"),
    ("output",   "attention.output.dense"),
    ("ffn_up",   "intermediate.dense"),                          #A
    ("ffn_down", "output.dense"),
])

for sub_name, sub_path in sublayer_paths.items():
    q_model = copy.deepcopy(model)
    for block in q_model.encoder.layer:
        layer = get_sublayer(block, sub_path)
        layer.weight.copy_(                                      #B
            fake_quantize_per_channel(layer.weight, bits))
    mse = F.mse_loss(
        q_model(**inputs).last_hidden_state, fp32_output).item()
```

Running on BERT-base-uncased:

Figure 5.15 Sublayer sensitivity to per-channel weight fake quantization on BERT-base-uncased. FFN sublayers dominate—16.2× more aggregate output MSE than all four attention projections at INT8, 2.0× at INT4. Dashed line separates attention (below) from FFN (above).



FFN layers produce the dominant weight quantization error at both precisions. This might seem surprising given Chapter 3's emphasis on attention sensitivity, but the distinction matters: Chapter 3 analyzed *activation* quantization—outlier channels and dynamic ranges in the values flowing through the network. Here we are quantizing *weights*, which are static and shaped by training. FFN weight matrices are larger (768×3072 versus 768×768 for attention projections) with wider per-channel magnitude ranges, producing more clipping under quantization. Attention projection weights are relatively smooth because the downstream softmax normalization constrains how extreme they need to be.

The diagnostic suggests a strategy: if you must leave some layers in higher precision, protect FFN first.

Apply QAT with selective strategies

Section 5.1 showed that INT4 PTQ collapses BERT-base on SST-2 to 50.9%—coin-flip accuracy on a binary task. The model is dead. Can QAT recover it, and does the sublayer ranking predict which selective strategy works best?

We wrap target layers with a fake-quantized replacement—the same pattern as FQConv2d from Section 5.4, adapted for `nn.Linear`:

Listing 5.16 Fake-quantized linear layer for transformer QAT

```
class FakeQuantizedLinear(nn.Module):
    """Drop-in replacement with per-channel fake quantization + S
    Same pattern as FQConv2d (Listing 5.13), adapted for nn.Linea

    def __init__(self, original_linear, bits=8):
        super().__init__()
        self.linear = original_linear                          #A
        self.bits = bits
        self.q_max = (1 << (bits - 1)) - 1

    def forward(self, x):
        w = self.linear.weight
        abs_max = w.detach().reshape(w.shape[0], -1).abs().amax(d
        scales = abs_max / self.q_max
        scales = torch.where(
            scales == 0, torch.ones_like(scales), scales)
        scales_bc = scales.view(-1, 1)

        w_q = torch.clamp(
            torch.round(w / scales_bc), -self.q_max, self.q_max)
        w_dq = w_q * scales_bc
        w_fq = w + (w_dq - w).detach()                         #B

        return F.linear(x, w_fq, self.linear.bias)
```

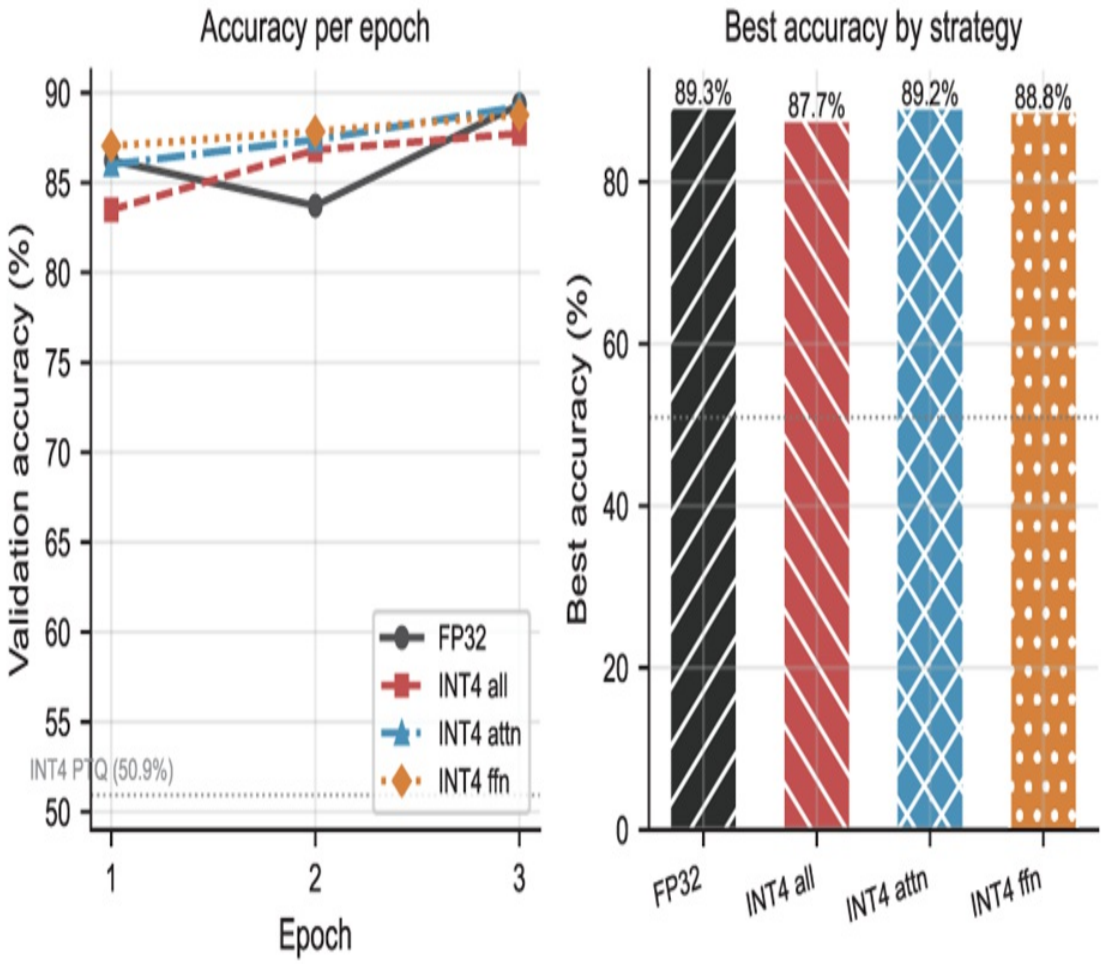
Three selective strategies: attention-only wraps Q, K, V, and output projections (48 layers); FFN-only wraps up and down projections (24 layers); all wraps every linear layer (72 total).

Listing 5.17 Selective QAT application to BERT

```
def apply_selective_qat(model, strategy="all", bits=8):
    count = 0
    for block in model.bert.encoder.layer:
        if strategy in ("all", "attn_only"):
            attn = block.attention
            attn.self.query = FakeQuantizedLinear(
                attn.self.query, bits)
            attn.self.key = FakeQuantizedLinear(
                attn.self.key, bits)
            attn.self.value = FakeQuantizedLinear(
                attn.self.value, bits)
            attn.output.dense = FakeQuantizedLinear(
                attn.output.dense, bits)
            count += 4
        if strategy in ("all", "ffn_only"):
            block.intermediate.dense = FakeQuantizedLinear(    #A
                block.intermediate.dense, bits)
            block.output.dense = FakeQuantizedLinear(
                block.output.dense, bits)
            count += 2
    return count
```

We fine-tune BERT-base-uncased on SST-2 for 3 epochs with each strategy, using a 2,000-sample training subset and the full 872-sample validation set. Every strategy starts from the same pretrained checkpoint with a randomly initialized classification head—same seed, same optimizer (AdamW, lr=2×10^{-5}), same data. Figure 5.16 shows accuracy per epoch for each strategy alongside the final best-accuracy comparison.

Figure 5.16 Selective QAT fine-tuning on SST-2 (INT4, 3 epochs). Left: accuracy per epoch. Right: best accuracy by strategy. QAT recovers BERT from the INT4 PTQ catastrophe (50.9%) to within 1.6% of FP32. Attention-only (48 layers at INT4) nearly matches FP32 at just −0.11%. FFN-only (24 layers) costs −0.57%. Quantizing all 72 layers incurs the largest gap (−1.61%).



First, QAT works. INT4 PTQ destroys the model—50.9% is random guessing on a binary task. With QAT, even the worst strategy recovers to 87.73%, and the best reaches 89.22%. The STE gives the optimizer enough gradient signal to learn around the quantization grid.

Second, the sensitivity diagnostic correctly predicted the strategy ranking. Attention projections, which produced the least weight distortion in Figure 5.15, are the easiest to train through—48 fake-quantized layers at just 0.11% cost. FFN layers, which dominated the distortion profile, cost 0.57% despite quantizing fewer layers. Quantizing all 72 layers accumulates noise from both groups for the largest penalty. A five-minute forward-pass diagnostic saved hours of trial-and-error.

Third, the gap between strategies is modest—1.5 points separates best from worst with only 3 epochs and 2,000 training samples. With more data and

training, these would converge further. The practical implication isn't that you must use attention-only; it's that when you're under a tight accuracy budget, the diagnostic tells you where the cheapest quantization capacity lies.

FP32 holdouts

LayerNorm parameters (γ and β) are absent from both the diagnostic and the QAT strategies. Each LayerNorm contributes two vectors of size 768—1,536 parameters versus 2.4 million in a single FFN layer. The compression savings are negligible, but LayerNorm's normalization role is critical to training stability. Softmax and GELU stay in FP32 for the same reason—they transform distributions, not parameters. Every production framework keeps these in FP32 by default.

Connecting to LLM scale

These principles—protect FFN weights, keep normalization in FP32—still apply at LLM scale, but larger models add a complication BERT does not exhibit. Dettmers et al. (2022) found that around 6.7B parameters, activation outliers coordinate across all transformer layers: the same handful of hidden dimensions become extreme outliers everywhere, and removing them degrades perplexity by 600–1000%. At that scale, the *activation* outlier problem dominates, and weight-only QAT of the kind built in this chapter is no longer sufficient on its own—activation-outlier-aware techniques such as LLM.int8(), SmoothQuant, and AWQ enter the picture.

Practical guidelines

Always keep LayerNorm, softmax, and GELU in FP32. Use per-channel fake quantization for all linear layers (axis 0, same as Section 5.4). At INT8, quantize everything—the diagnostic shows cosine similarity above 0.999 for all sublayers. At INT4, start with attention projections (cheapest to quantize), add FFN after warmup—or keep FFN at INT8 in a mixed-precision configuration. Run the sublayer diagnostic on your specific architecture before choosing a strategy.

The companion script `ch5/ch5_transformer_qat.py` implements both experiments:

```
# Quick sublayer diagnostic (Figure 5.15)
python ch5_transformer_qat.py --mode sensitivity --save-plots

# Selective QAT fine-tuning on SST-2 (Figure 5.16)
python ch5_transformer_qat.py --mode qat --save-plots

# Run everything
python ch5_transformer_qat.py --mode all --save-plots
```

5.6 Summary

- QAT is warranted when PTQ fails at your target bit-width, but the failure pattern determines the intervention. INT4 PTQ dropped ResNet-18 by 12 percentage points (diffuse, across all layers) and collapsed BERT-base to 50.9% coin-flip accuracy (concentrated in a single FFN layer). Diffuse failure needs QAT; concentrated failure needs mixed precision. Always run layer sensitivity before choosing.
- Fake quantization inserts quantize-then-dequantize operations into the forward pass so the model trains against quantization noise while gradients flow through the straight-through estimator. The clipped STE —which zeros gradients for out-of-range values—is the recommended default. Use lower learning rates (1e-4 to 1e-5), shorter schedules (5–20 epochs), and gradient clipping.
- Scheduling reduces QAT cost without sacrificing accuracy. Warm up the learning rate for 10–15% of total epochs, freeze observers once per-layer scale changes drop below 1%, and progress bit-width gradually (INT8 → INT6 → INT4). Our scheduled pipeline reached 90.30% on SmallCNN at INT4—slightly above the FP32 baseline—with the same wall-clock time as naïve QAT.
- Per-channel fake quantization prevents gradient starvation during QAT. Per-tensor INT4 quantization clips 100% of gradients in some output channels; per-channel keeps all channels below 20%. Quantize along axis 0 (output channels) for both convolutions and linear layers.
- Transformer QAT requires selective application. FFN sublayers produce 16.2× more aggregate output MSE than attention projections at INT8.

Attention-only QAT recovered BERT from 50.9% to within 0.11% of FP32; quantizing all layers incurred 1.61%. Always keep LayerNorm, softmax, and GELU in FP32—their parameter count is negligible but their numerical sensitivity is high.

- When diminishing returns set in, stop. Monitor observer ΔScale and validation accuracy jointly—when per-layer scale changes flatten below 1% and accuracy stops recovering after bit-width transitions, further epochs add cost without benefit.

6 Porting Workflows Across Frameworks

This chapter covers

- Recognizing where quantization workflows diverge across frameworks
- Running end-to-end quantization in PyTorch and TorchAO
- Exporting to ONNX with correct quantization semantics
- Converting to TensorFlow and TFLite quantized formats
- Validating numerical equivalence across exported artifacts

Up to this point, we've discussed quantization within a single framework. You learned to choose granularity, calibrate ranges, and recover accuracy with QATin PyTorch. But production teams rarely live in a single framework: A research team trains in PyTorch, the serving team needs an ONNX artifact (a portable model graph in an open exchange format that we'll cover in detail in section 6.4), and the mobile team needs TensorFlow Lite. Same model, same quantization intent, three different export targets.

This is where quantization projects run into trouble. You quantize in PyTorch, export to ONNX, and accuracy drops 1.3%, not because either tool has a bug, because each framework makes different default choices about how to quantize, and those choices compound.

That 1.3% matters more than it looks.

Consider a product recommendation model serving 50 million queries a day. A 1.3% accuracy drop means 650,000 additional wrong predictions daily—degraded recommendations that erode user trust, reduce click-through, and are nearly impossible to trace back because each step in the pipeline appears correct on its own. The PyTorch model passes its tests. The ONNX export passes its tests. The serving stack is working as designed. The bug lives in the gap between them: disagreements about how quantization is implemented.

The fix isn't to standardize on one framework—that's rarely possible. The fix is to understand exactly where the paths diverge, produce artifacts through each one deliberately, and verify that they agree before anything reaches production.

This is where deployment work becomes less about training and more about consistency. If you’re responsible for moving models between frameworks—or ensuring they behave the same once they get there—these differences aren’t edge cases; they’re part of the job.

6.1 Know where frameworks diverge

You've quantized a model in PyTorch. Accuracy is within budget, latency meets the target, Continuous Integration (CI) is green. Then someone exports the same model to ONNX for the serving stack, and top-1 accuracy is off by 0.8%. A handful of edge-case inputs flip predictions. The deployment team files a bug. The research team can't reproduce it. Both teams are correct.

This scenario plays out routinely in organizations where training and serving span different stacks. It's not caused by a defect in any single tool. It's caused by the fact that "INT8 quantization" is not a single algorithm - it's a family of choices, and each framework makes those choices differently.

What actually diverges

First, default quantization semantics differ. PyTorch's eager-mode quantization defaults to per-tensor symmetric for weights and per-tensor asymmetric for activations. TensorFlow's quantization-aware training defaults to per-axis (per-channel) symmetric for weights. ONNX Runtime's static quantization defaults to per-tensor symmetric for both. Even when you specify "INT8 static quantization" in all three, the resulting quantization grids are not identical because the default granularity, observer algorithms, and range estimation methods vary.

Second, operator fusion and graph rewriting rules differ. Operator fusion is a compiler optimization where the framework merges adjacent operations—say, a convolution followed by batch normalization followed by ReLU—into

a single combined operation that avoids writing intermediate results to memory. PyTorch fuses Conv-BN-ReLU into a single quantized operator with specific rounding behavior. TensorFlow performs batch norm folding during conversion with its own rounding semantics. ONNX Runtime's graph optimizer applies yet another set of fusion patterns. The fused operators aren't numerically identical - they agree to within floating-point tolerance on most inputs, but outlier activations can diverge meaningfully.

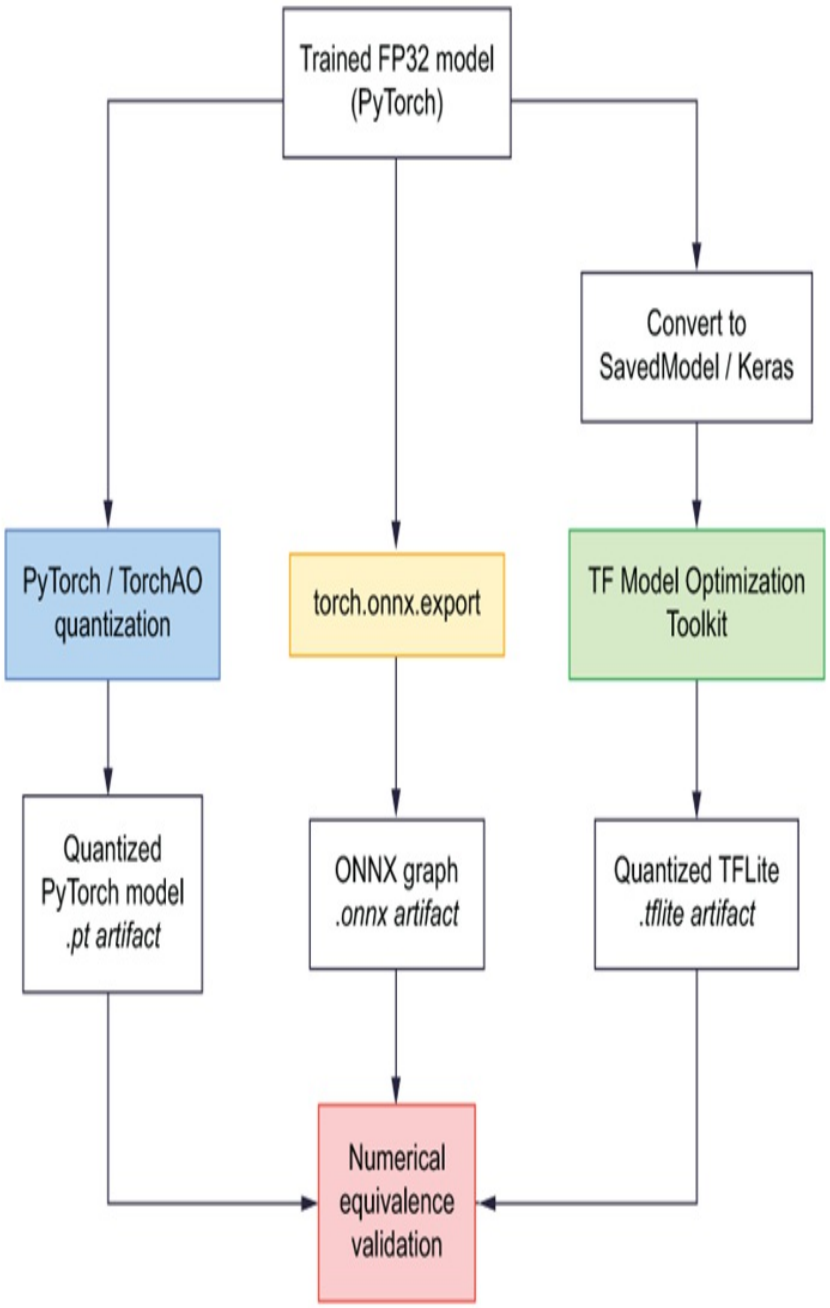
Third, calibration implementations produce different ranges. Even when two frameworks nominally use "min-max" calibration, implementation details matter: how running statistics are accumulated, whether the min-max window is per-batch or global, how ties are broken in histogram-based methods. A 0.1% difference in computed scale propagates through every matmul in the network.

The parity landscape at a glance

The divergence points - where each framework makes independent quantization decisions - are where parity breaks.

Figure 6.1 shows the typical flow from a trained FP32 model to quantized artifacts across the three major paths.

Figure 6.1 Three quantization paths from a trained PyTorch model to deployment-ready artifacts. Arrows represent conversion steps: downward arrows feed the FP32 model into each framework's quantization pipeline, horizontal arrows carry the model through quantization and optimization stages, and converging arrows at the bottom feed all three artifacts into a shared validation step. Each path makes independent decisions about quantization semantics, operator fusion, and calibration.



Reach doesn't mean bitwise-identical outputs - floating-point arithmetic isn't associative, so reordering operations (which every graph optimizer does) changes the least significant bits.

Achieving means three things: predictions agree on the same inputs (no label flips on your test set), accuracy metrics are within a documented tolerance (typically <0.5% top-1 for classification, <1.0 perplexity for language

models), and the differences are explainable by known quantization-semantic choices rather than export bugs.

So why can't you just pick one format and ignore the rest?

Pragmatically, teams end up crossing framework boundaries for three reasons. First, the best quantization algorithm for your model may not exist in your target format - PyTorch's FX graph mode quantization offers calibration strategies that ONNX doesn't natively represent, while ONNX's operator set enables optimizations that PyTorch's eager mode can't express. Second, hardware targets force the issue: TensorRT consumes ONNX, TFLite requires TensorFlow conversion, and CoreML has its own quantization pipeline. Third, CI and validation infrastructure often lives in a different framework than the serving stack - your accuracy regression tests may run in PyTorch while the artifact shipped to production is an ONNX file.

6.2 Run the PyTorch and TorchAO path end-to-end

Now that we've laid out three places where frameworks diverge, we need concrete artifacts to compare differences. This section builds the first set—quantized models through PyTorch's own tooling—so we have a baseline to measure the TensorFlow and ONNX paths against.

We apply TorchAO's `quantize_()` API to three models — a CNN, a transformer, and an LLM — and measure exactly what one line of quantization code delivers: how much smaller, how accurate, and how fast.

TorchAO's `quantize_()` walks the module tree, finds every `nn.Linear`, and replaces its weight tensor with an `AffineQuantizedTensor` subclass that stores INT8 data plus per-channel scales. The module stays as `nn.Linear` — no structural changes, no graph capture, no calibration data for weight-only configs.

Predict compression before writing code

Whether `quantize_()` helps depends on where parameters live. Table 6.1 shows the architecture breakdown for our three models.

Table 6.1 Architecture determines compression ceiling

Model	Linear layers	Linear params (%)	Theoretical INT8 ceiling	Measured
ResNet-18	1	513,000 (4.4%)	1.03×	1.03×
BERT-base	74	85,609,730 (78.2%)	2.42×	2.40×
TinyLlama-1.1B	155	1,034,420,224 (94.0%)	3.57×	3.38×

The ceiling column is arithmetic you can do before touching the quantization code. If `p` is the fraction of parameters in `nn.Linear`, and INT8 compresses those 4× while everything else stays in FP32, the maximum compression is *1 / (p × 0.25 + (1 − p) × 1.0)*. For BERT: *1 / (0.782 × 0.25 + 0.218 × 1.0) = 2.42*×.

The measured 2.40× lands within 1% of the theoretical limit.

For TinyLlama, the gap is larger (3.38× vs 3.57×) because `AffineQuantizedTensor` stores per-channel FP32 scales alongside the INT8 data — overhead that grows with the number of layers.

Listing 6.1 shows the parameter count that produces Table 6.1.

Listing 6.1 Count parameters by layer type to predict compression

```
n_linear = sum(1 for m in model.modules()
               if isinstance(m, nn.Linear))
p_linear = sum(p.numel() for m in model.modules()
               if isinstance(m, nn.Linear)
               for p in m.parameters())
total = sum(p.numel() for p in model.parameters())
print(f"Linear: {n_linear} layers, {p_linear/total:.1%} of params
```

ResNet-18: the architectural lesson

ResNet-18 has 20 Conv2d layers holding 95.5% of its parameters and a single `nn.Linear` classification head. `quantize_()` converts that one layer

and leaves everything else untouched:

```
quantize_(resnet, Int8WeightOnlyConfig())
# → 1/1 nn.Linear layers quantized
# → 43.22 MB (1.03× compression)
```

This is not a bug. TorchAO's stable quantization configs (`Int8WeightOnlyConfig`, `Int8DynamicActivationInt8WeightConfig`) internally reshape weights to 2D via `block_size = (1, weight.shape[-1])`.

Conv2d weights are 4D tensors — they fail this reshape. TorchAO was designed for transformer workloads where `nn.Linear` dominates. CNN quantization in PyTorch requires a different API (PT2E export quantization with a backend-specific quantizer like XNNPACK), which we cover alongside runtime deployment in Chapter 9.

BERT-base on SST-2: the transformer sweet spot

BERT-base has 74 `nn.Linear` layers holding 78.2% of its parameters. This is the architecture TorchAO was built for.

Listing 6.2 TorchAO quantization in one line

```
from torchao.quantization import quantize_, Int8WeightOnlyConfig

model = AutoModelForSequenceClassification.from_pretrained(
    "textattack/bert-base-uncased-SST-2"
)
quantize_(model, Int8WeightOnlyConfig())
```

After this call, inspection confirms what changed. The first three layers of encoder block 0:

```
bert.encoder.layer.0.attention.self.query:
    AffineQuantizedTensor, shape=[768, 768], block_size=(1, 768),
    int_dtype=torch.int8
bert.encoder.layer.0.attention.self.key:
    AffineQuantizedTensor, shape=[768, 768], block_size=(1, 768),
    int_dtype=torch.int8
```

The `block_size=(1, 768)` confirms per-channel quantization — one scale per output row, matching the granularity analysis from Chapter 3. All 74 linear layers are quantized; the word embedding (23.4M parameters, 21.4% of the model) stays in FP32 because it is `nn.Embedding`, not `nn.Linear`.

Table 6.2 shows the results across two configs.

Table 6.2 BERT-base on SST-2 (872 validation samples, T4 GPU)

Config	Size	Compression	Accuracy	Δ Accuracy
FP32 baseline	417.73 MB	1.00×	92.43%	—
INT8 weight-only	174.05 MB	2.40×	92.32%	-0.11%
W8A8 dynamic	173.40 MB	2.41×	92.20%	-0.23%

Two important observations to take note of:

First, accuracy loss is negligible — 0.11% on a binary classification task, well within noise. INT8 per-channel quantization on well-trained transformer weights rarely causes measurable degradation, consistent with what Chapter 4 established for PTQ.

Second, both configs achieve nearly identical compression (2.40× vs 2.41×) because the storage savings come from INT8 weights, not from activation quantization. W8A8 dynamic quantizes activations at inference time — a compute-time operation that does not reduce the saved model size. The W8A8 config exists for deployment runtimes that can exploit both quantized weights and quantized activations for faster integer arithmetic.

In eager-mode PyTorch, it adds overhead without benefit.

TinyLlama-1.1B: compression at scale

The storage argument becomes tangible at LLM scale. TinyLlama-1.1B has 155 linear layers holding 94% of its 1.1 billion parameters.

```
quantize_(tinyllama, Int8WeightOnlyConfig())
```

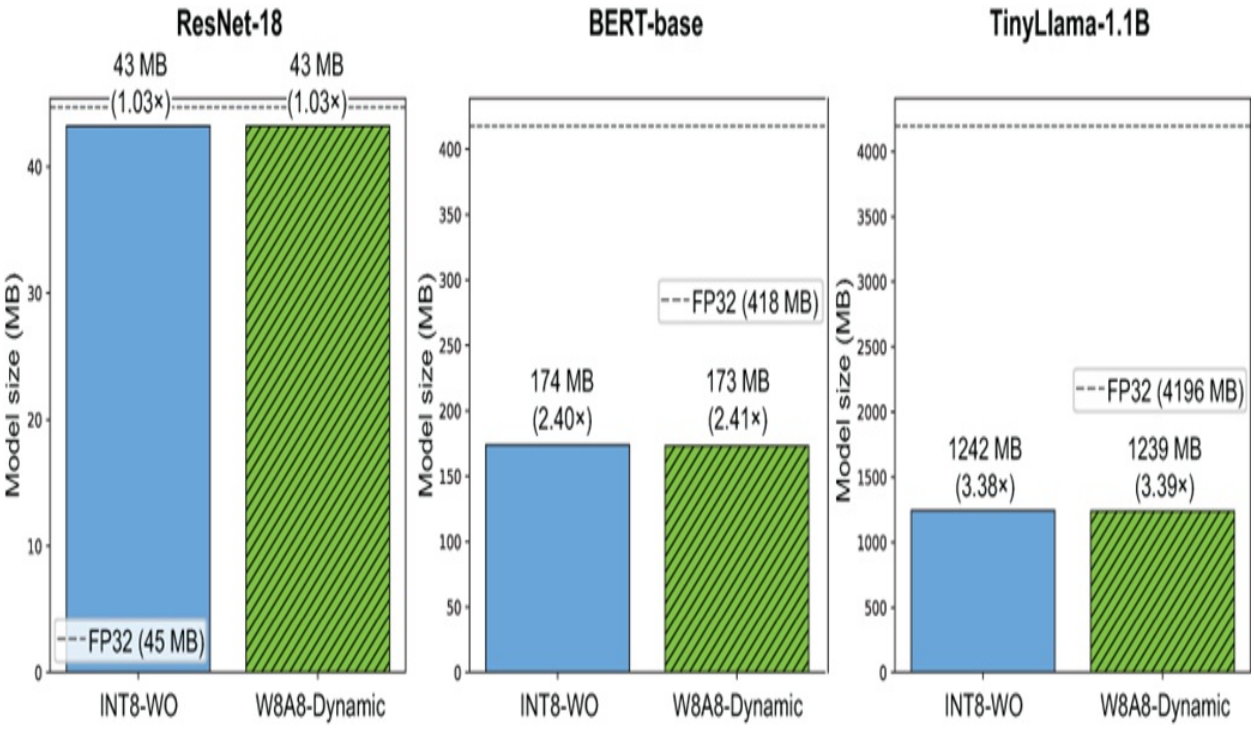
```
# → 155/155 nn.Linear layers quantized
# → 1,241.93 MB (3.38× compression)
```

That is a 4.2 GB model fitting in 1.2 GB. In production, this means fitting on a smaller GPU (a T4 with 16 GB instead of needing headroom for the full FP32 checkpoint), 3× more model replicas per node for throughput scaling, and ~3 GB less to download and distribute per deployment.

The accuracy impact requires perplexity evaluation on a language modeling benchmark — perplexity measures how well a model predicts held-out text, with lower values indicating better predictions.

Figure 6.2 shows the compression results side by side across all three models.

Figure 6.2 TorchAO INT8 weight-only compression across three architectures. Compression scales with the fraction of parameters in nn.Linear: 1.03× for the Conv2d-dominated ResNet-18, 2.40× for BERT-base, and 3.38× for TinyLlama-1.1B.



NOTE

Quantization reduces model size unconditionally. It does not reduce inference latency in eager-mode PyTorch.

Table 6.3 shows the measured latencies.

Table 6.3 Eager-mode latency after TorchAO quantization (T4 GPU)

Model	Config	Latency	vs FP32
BERT-base	FP32 baseline	9.33 ms	1.00×
BERT-base	INT8-WO	17.69 ms	0.53×
TinyLlama-1.1B	FP32 baseline	60.87 ms	1.00×
TinyLlama-1.1B	INT8-WO	85.07 ms	0.72×

Every quantized model is slower than its FP32 baseline. This is expected and not a defect. `quantize_()` replaces weight tensors with `AffineQuantizedTensor` subclasses. At each matmul, PyTorch dequantizes the INT8 weights back to FP32 and runs the standard FP32 kernel — strictly more work than the original. The weights are stored smaller, but computation still happens in floating point.

Latency improvement requires a runtime that can execute integer arithmetic natively: ONNX Runtime, TensorRT, OpenVINO, or `torch.compile` with fused quantized kernels on supported hardware. Those deployment paths are the subject of Chapter 9. What this section delivers — a correctly quantized, accuracy-verified, serializable artifact — is the input that those runtimes consume.

Serialize and reload

TorchAO models serialize through `state_dict`, not full-model pickling. The `AffineQuantizedTensor` subclasses survive the round-trip when you reconstruct the quantized skeleton first.

Listing 6.3 Save/load round-trip for TorchAO models

```
# Save
torch.save(model.state_dict(), "bert_int8wo.pt")

# Load
```

```
loaded = AutoModelForSequenceClassification.from_pretrained(...)
quantize_(loaded, Int8WeightOnlyConfig())
loaded.load_state_dict(
    torch.load("bert_int8wo.pt", weights_only=False)
)
```

The round-trip preserves accuracy exactly: 92.32% before save, 92.32% after load on SST-2. The serialized file is 174.05 MB — the same 2.40× compression as the in-memory model.

Full-model pickling (`torch.save(model)`) can fail across PyTorch versions because internal module attributes change between releases; state-dict is the portable path.

When to use TorchAO quantize_()

The pattern across all three models points to a simple rule: if most of your model's parameters live in nn.Linear layers, TorchAO's one-line API delivers meaningful compression with negligible accuracy cost. If they live in Conv2d layers, you need a different tool. Table 6.4 captures this decision.

Table 6.4 TorchAO quantize_() decision rule

Your model	Linear params	Expected compression	Use TorchAO?
Transformer (BERT, GPT, LLaMA)	>70%	2–4×	Yes
CNN (ResNet, EfficientNet, YOLO)	<10%	~1×	No — use PT2E or runtime quantizer (Chapter 9)
Hybrid (ViT, ConvNeXt)	Varies	Compute ceiling first	If Linear-dominant, yes

The one-line API, zero-calibration weight-only config, and state-dict serialization make `quantize_()` the fastest path from a trained transformer to a compressed artifact. It is not the path to faster inference — that requires a runtime. But for teams that need to reduce storage, cut download sizes, or fit

larger models on smaller hardware, compression alone pays for itself.

(Full experiment code: ch6/ch6_pytorch_torchao_path.py `--all --save-plots`*)*

6.3 Run the TensorFlow path with the model optimization toolkit

With TorchAO artifacts in hand, we have one side of the cross-framework comparison. But many production teams don't deploy through PyTorch—mobile apps, edge devices, and on-device ML pipelines typically run TensorFlow Lite. To test whether our quantization results hold across the framework boundary, we now run the same architectures through TensorFlow's post-training quantization pipeline, accessed via the TFLite converter. The results expose exactly the kind of divergence that section 6.1 warned about.

Two models run the TF path. MobileNetV2 replaces ResNet-18 as the vision model because it is the canonical TFLite deployment target — purpose-built for mobile, with depthwise separable convolutions that TFLite's INT8 kernels are optimized for. BERT-base on SST-2 carries over directly, loaded from the same HuggingFace checkpoint via `from_pt=True` to convert PyTorch weights into TensorFlow format.

Three lines of configuration, three different artifacts

TFLite post-training quantization is controlled entirely through the `TFLiteConverter`. The mode is determined by which flags you set before calling `convert()`. Listing 6.4 shows the three configurations we apply.

Listing 6.4 TFLite converter configurations

```
import tensorflow as tf
converter = tf.lite.TFLiteConverter.from_saved_model(saved_model_

# Mode 1: Dynamic range — weights to INT8, no calibration
converter.optimizations = [tf.lite.Optimize.DEFAULT]
```

```
# Mode 2: Full integer — weights AND activations to INT8
converter.optimizations = [tf.lite.Optimize.DEFAULT]
converter.representative_dataset = representative_dataset_fn

# Mode 3: Float16 — weights stored as FP16
converter.optimizations = [tf.lite.Optimize.DEFAULT]
converter.target_spec.supported_types = [tf.float16]

tflite_model = converter.convert()
```

The API is simpler than TorchAO's because the converter handles everything in a single pass: graph optimization, operator fusion, calibration, and serialization. There is no separate `quantize_()` step followed by manual `torch.save()`. The cost of that simplicity is less control — you cannot selectively quantize individual layers or swap observers the way PyTorch's PT2E API allows.

MobileNetV2: the TFLite sweet spot

MobileNetV2 has 3.5 million trainable parameters. Its architecture is 62.5% Conv2d and DepthwiseConv2d, 36.5% Dense, and 1.0% other (batch normalization, with gamma and beta as trainable weights). This is the opposite of the ResNet-18 story from section 6.2, where 95.5% of parameters lived in Conv2d and TorchAO's Linear-only quantization gave a useless 1.03×. Here, the TFLite converter quantizes *all* layer types — Conv2d, DepthwiseConv2d, and Dense — so nearly every parameter is eligible.

Table 6.5 shows the results across all four configurations on 3,925 ImageNette validation samples.

Table 6.5 MobileNetV2 on ImageNette (3,925 validation samples, T4 GPU)

Config	Size (MB)	Compression	Accuracy	Δ Accuracy (percentage points)
FP32 (baseline)	13.34	1.00×	84.82%	—
Dynamic range	3.62	3.68×	83.54%	−1.28 pp

Full integer	3.83	3.48×	83.24%	−1.58 pp
INT8 I/O (edge)	3.83	3.48×	83.24%	−1.58 pp
Float16	6.70	1.99×	84.94%	+0.12 pp

Examining Table 6.5, we can come away with three immediate observations.

First, dynamic range achieves 3.68× compression with only 1.28 percentage points of accuracy loss — and no calibration data required. This is the simplest TFLite quantization path: one flag (`Optimize.DEFAULT`), one call, done.

Second, full integer quantization gives slightly worse compression (3.48×) than dynamic range (3.68×) despite quantizing both weights and activations. The model is *larger* because the converter adds per-tensor quantization parameters (scales and zero-points) for every activation tensor — metadata overhead that outweighs the activation savings on a model this small. The accuracy cost is modest (−1.58 pp), confirming that CNN activations — bounded by ReLU6 to [0, 6] — are stable enough for static range calibration.

Third, FP16 is essentially free. The +0.12 pp accuracy improvement is noise — the SST-2 validation set has 872 samples, so 0.12 pp is a single-sample fluctuation well within run-to-run variance — and the 1.99× compression comes from halving every weight from 32 bits to 16 bits. No calibration, no accuracy risk, identical throughput. For teams that need to shrink a model without touching accuracy, FP16 is the zero-risk option.

The representative dataset pattern

Full integer quantization requires a representative dataset — a Python generator that yields sample inputs. Listing 6.5 shows the pattern for MobileNetV2.

Listing 6.5 Representative dataset generator for calibration

```
def make_representative_dataset(images, num_calibration=200):
    def generator():
        indices = np.random.RandomState(42).choice(
```

```
            len(images), size=min(num_calibration, len(images)),
            replace=False
        )
        for i in indices:
            yield [images[i:i+1].astype(np.float32)]
    return generator
```

This serves the same role as the calibration set from Chapter 4, but the mechanism is different. In PyTorch's PT2E, calibration runs through an explicit `prepare_pt2e` → forward pass → `convert_pt2e` pipeline where you control the calibration loop.

In TFLite, the converter owns the calibration loop internally — you hand it a generator and it handles the rest. The tradeoff: less code, but no visibility into which layers received which ranges or how outliers were handled.

Two hundred calibration samples is sufficient for most vision models. More samples tighten the activation ranges but with diminishing returns beyond ~200. For models with high activation variance (transformers), static calibration is the wrong approach entirely — which brings us to BERT.

BERT-base: dynamic range only

BERT-base on SST-2 is loaded from the same HuggingFace checkpoint as section 6.2, but via TensorFlow's backend (`TFAutoModelForSequenceClassification` with `from_pt=True`). The weight conversion transposes and remaps parameter tensors between PyTorch and TF conventions. On SST-2, the FP32 TFLite model scores 92.43% — matching the FP32 PyTorch baseline from section 6.2 exactly (806/872 correct).

The cross-framework weight conversion did not change a single prediction at full precision. Divergence appears only after quantization.

One ecosystem constraint to be aware of: HuggingFace's `transformers` library dropped TensorFlow model classes in version 5.0 (2025). Loading models via `TFAutoModelForSequenceClassification` requires pinning to `transformers<5.0`. This does not affect TFLite conversion of native Keras models (MobileNetV2, EfficientNet, etc.) — only the cross-framework

loading path from PyTorch checkpoints via HuggingFace.

A critical difference from PyTorch: TFLite requires *fixed input shapes*. The SavedModel must be traced with concrete tensor specifications before the converter will accept it:

```
@tf.function(input_signature=[
    tf.TensorSpec(shape=[1, 128], dtype=tf.int32, name="input_ids
    tf.TensorSpec(shape=[1, 128], dtype=tf.int32, name="attention
])
def serving_fn(input_ids, attention_mask):
    return {"logits": model(input_ids=input_ids,
                            attention_mask=attention_mask).logits
```

This pins batch size to 1 and sequence length to 128. TorchAO's `quantize_()` works in eager mode without shape specialization — a usability difference that matters when your serving infrastructure handles variable-length inputs.

Table 6.6 shows the BERT results. Only dynamic range and FP16 are shown — full integer quantization is omitted for transformers.

Table 6.6 BERT-base on SST-2 via TFLite (872 validation samples, T4 GPU)

Config	Size (MB)	Compression	Accuracy	Δ Accuracy
FP32 (baseline)	416.65	1.00×	92.43%	—
Dynamic range	105.85	3.94×	92.43%	+0.00 pp
Float16	208.41	2.00×	92.43%	+0.00 pp

Dynamic range quantization gives 3.94× compression with zero accuracy loss on 872 validation samples. The logits shift by small amounts (typically 0.01–0.05) but never enough to flip a prediction on this binary task.

This matches TorchAO's INT8 weight-only result from section 6.2 (92.32% / 2.40×), though the compression is better here because the TFLite FP32 baseline is larger (416.65 MB vs TorchAO's 174 MB — the TFLite flatbuffer includes additional graph metadata).

Why full integer is omitted for BERT

Full integer quantization applies *static* INT8 scales to activations — computed once during calibration and baked into the model. MobileNetV2's activations are bounded by ReLU6, so static ranges work.

Transformer activations — layer norm outputs, attention scores after softmax, GELU intermediates — vary dramatically across inputs. A single per-tensor scale calibrated on 200 samples cannot faithfully represent the full activation distribution. On BERT, full integer quantization produces logits near zero (effectively random) and collapses to 50% accuracy on the binary SST-2 task. Dynamic range quantization avoids this by computing activation scales at inference time from the actual input — the correct TFLite path for any transformer model.

Cross-framework comparison

Table 6.7 shows the same quantization intent — INT8 weights, no calibration — applied through both frameworks. The accuracy and compression differences are the parity gaps from section 6.1.

Table 6.7 Cross-framework comparison: INT8 weight-only quantization

Model	Framework	API	Compression	Accura
ResNet-18	TorchAO	`Int8WeightOnlyConfig`	1.03×	—
MobileNetV2	TFLite	Dynamic range	3.68×	83.54%
BERT-base	TorchAO	`Int8WeightOnlyConfig`	2.40×	92.32%
BERT-base	TFLite	Dynamic range	3.94×	92.43%

The ResNet-18 vs MobileNetV2 gap is architectural, not framework-related — different models, different parameter distributions.

The BERT comparison is more revealing: the same checkpoint quantized through two frameworks gives 2.40× (TorchAO) vs 3.94× (TFLite) compression and 92.32% vs 92.43% accuracy.

The size difference is partly format overhead (TFLite flatbuffer vs PyTorch state-dict) and partly because TFLite's dynamic range quantizes all weight tensors including embeddings, while TorchAO's `Int8WeightOnlyConfig` only targets `nn.Linear`. The 0.11 percentage point accuracy difference is within noise for 872 binary classification samples — the models functionally agree.

A note on TFLite naming

TensorFlow Lite was rebranded to LiteRT (Lite Runtime) in late 2024 as part of Google's AI Edge suite. The Python API (`tf.lite.TFLiteConverter`, `tf.lite.Interpreter`) remains unchanged as of TF 2.18, though `tf.lite.Interpreter` prints a deprecation warning pointing to the `ai_edge_litert` package. The `.tflite` file format is unchanged. If you see "LiteRT" in documentation, it refers to the same runtime and the same conversion pipeline described here.

When to use TFLite quantization

The TFLite results point to a clear pattern: dynamic range quantization is the default choice for transformers because it avoids the activation-range problems that break full integer quantization on attention-heavy architectures. Full integer is the right path for CNNs bound for mobile or edge hardware, where ReLU-bounded activations are stable enough for static ranges. FP16 is the zero-risk option when you need compression but cannot afford any accuracy change. Table 6.8 captures these decision rules.

Table 6.8 TFLite quantization decision guide

Scenario	Config	Why
Transformer, need compression	Dynamic range	Zero accuracy loss, no calibration

CNN, mobile/edge deployment	Full integer	Static INT8 for NNAPI, Coral, XNNPACK
CNN, integer-only hardware	INT8-strict (I/O)	Microcontrollers, Edge TPU
Any model, minimal risk	Float16	2× compression, no accuracy cost
Transformer, need W8A8	Not TFLite — use TorchAO W8A8 dynamic	TFLite's full-integer uses static activation ranges

(Full experiment code: ch6/ch6_tf_mot_path.py *--all --mode all --save-plots)*

6.4 Export to ONNX and drive ONNX Runtime quantization

Sections 6.2 and 6.3 quantized models inside the framework they were trained in. TorchAO replaced weight tensors in-place; TFLite converted a SavedModel and applied quantization in the same pass.

Both paths assume the quantization tool and the deployment runtime share a framework. In production, they often don't. A model trained in PyTorch may need to run in a C++ inference server, an embedded device, or a cloud endpoint backed by hardware the training framework has never seen.

That gap is what ONNX exists to close.

Understand what ONNX is and why it matters

ONNX - Open Neural Network Exchange - is an open-source format for representing machine learning models as portable computation graphs. Microsoft and Facebook (now Meta) launched it in September 2017 to solve a concrete problem: a model trained in one framework could not be deployed in another without rewriting the model code. A ResNet trained in PyTorch could not run on Intel's OpenVINO. A BERT fine-tuned in TensorFlow could not be served by Microsoft's inference stack. Every framework had its own graph representation, its own operator semantics, and its own serialization

format.

ONNX defines three things that make portability possible.

First, a graph format: each model is a directed acyclic graph of operator nodes, where every node takes named tensor inputs and produces named tensor outputs. Second, a standardized operator set (opset): over 180 operators - Conv, MatMul, Relu, Softmax, Reshape, and so on - with precise mathematical definitions that every runtime must implement identically. Third, a serialization format based on Protocol Buffers that encodes the graph structure, operator parameters, and weight tensors into a single `.onnx` file.

NOTE

ONNX is the format - the .onnx file. ONNX Runtime (ORT) is Microsoft's inference engine that executes those files. ORT is the dominant runtime, but it is not the only one. TensorRT reads ONNX graphs.

OpenVINO reads ONNX graphs. CoreML can import ONNX through converters. The format is the lingua franca; the runtimes are the interpreters. This section exports FP32 models from PyTorch to ONNX and applies ORT's quantization to produce artifacts comparable with the TorchAO and TFLite results from sections 6.2 and 6.3. Hardware-targeted optimization and the Optimum deployment API are the subject of section 9.2.

Export a PyTorch model to ONNX

The export step converts a live PyTorch `nn.Module` into an ONNX graph. PyTorch provides `torch.onnx.export` for this, and the function has two exporter backends controlled by the `dynamo` flag. The legacy path (`dynamo=False`) uses TorchScript: it traces the model with example inputs, records every operation, and writes the trace as an ONNX graph. This is the exporter that has shipped with PyTorch since 2017.

The newer path (`dynamo=True`) uses `torch.export`, PyTorch's graph-capture system introduced in PyTorch 2.0. It is built on TorchDynamo, a bytecode-level analysis tool that captures the computation graph more completely than

TorchScript tracing - handling more Python control flow and using less memory. The `dynamo` flag in `torch.onnx.export` is named after this underlying component.

As of PyTorch 2.9, `dynamo=True` is the default. The PyTorch team filed the prerequisite tracking issue (`#151693`) in April 2025, listing sub-issues that needed resolution before the default could safely flip - including control flow rewrites, fallback mechanism improvements, and test migration. The default flipped regardless, and the tracker shows failures on several model families afterward: GRU, YOLOv3-class architectures, `FakeQuantize` layers, and others. The legacy `dynamic_axes` argument also has incomplete support in the dynamo path; the dynamo exporter prefers the newer `dynamic_shapes` parameter.

In our case, the dynamo exporter failed on ResNet-18 with an opset version conversion error (18 to 17), producing a 0.09 MB file - a graph skeleton without weights that passed `onnx.checker.check_model` but would produce garbage at inference time. We use `dynamo=False` throughout this section. If your export produces unexpected results - opset conversion warnings, exceptions during graph capture, or outputs that don't match PyTorch - `dynamo=False` is the first thing to try.

Listing 6.6 shows the ResNet-18 export. The call is one line, but the arguments encode decisions that matter downstream.

Listing 6.6 Export ResNet-18 to ONNX

```
torch.onnx.export(
    model,
    dummy_input,                                          #A
    "resnet18_fp32.onnx",
    opset_version=17,                                     #B
    input_names=["input"],
    output_names=["output"],
    dynamic_axes={"input": {0: "batch"},
                  "output": {0: "batch"}},                #C
    dynamo=False,                                         #D
)
```

The exported ResNet-18 is 44.58 MB - the same 11.2 million FP32

parameters as the PyTorch model, now serialized in ONNX's protobuf format. Running it through ORT on CPU with the full ImageNette validation set gives 99.21% accuracy, matching the PyTorch baseline exactly. The export is lossless.

BERT-base requires more care. It has three inputs (input_ids, attention_mask, token_type_ids) and needs dynamic axes on both the batch and sequence length dimensions so the same ONNX file can handle inputs of any length. The TorchScript tracer handles HuggingFace's `SequenceClassifierOutput` dataclass by extracting the logits tensor automatically.

The exported BERT graph is 417.85 MB with 1,003 operator nodes - roughly 10× more operators than ResNet-18's 49 nodes, reflecting the depth of a 12-layer transformer.

Apply ORT's quantization to the exported graph

With valid FP32 ONNX files in hand, we apply ONNX Runtime's own quantization tools. ORT provides two functions in `onnxruntime.quantization: quantize_dynamic` for weight-only quantization with on-the-fly activation scaling, and `quantize_static` for full W8A8 quantization with calibration. These operate on the ONNX file directly - no PyTorch, no TensorFlow, no GPU required. The quantized output is another `.onnx` file.

ResNet-18: static quantization is the correct path

Dynamic quantization targets MatMul and Gemm operators by default. ResNet-18 has 20 Conv layers and a single Gemm (the classifier head), so dynamic quantization compresses only that one layer - 43.12 MB, a 1.03× reduction. This is the same result as TorchAO's `Int8WeightOnlyConfig` from section 6.2 and for the same structural reason: both tools only quantize the linear portion of the model.

Static quantization quantizes everything. It follows a three-step pipeline: preprocess the FP32 ONNX graph (shape inference and graph optimization), calibrate with representative data to collect activation ranges, and quantize by

inserting QuantizeLinear/DequantizeLinear (QDQ) node pairs throughout the graph.

Listing 6.7 shows the calibration data reader, which is ORT's equivalent of Chapter 4's calibration set.

Listing 6.7 Calibration data reader for ORT static quantization

```
class ResNetCalibrationReader(CalibrationDataReader):
    def __init__(self, images, input_name="input"):
        self.images = images
        self.input_name = input_name
        self.index = 0

    def get_next(self):                                    #A
        if self.index >= len(self.images):
            return None
        data = {self.input_name:
                self.images[self.index:self.index+1]}
        self.index += 1
        return data

    def rewind(self):
        self.index = 0
```

The static quantization call configures the quantization grid:

```
quantize_static(
    model_input=preprocessed_path,
    model_output="resnet18_static_int8.onnx",
    calibration_data_reader=calib_reader,
    quant_format=QuantFormat.QDQ,                          #A
    activation_type=QuantType.QUInt8,                      #B
    weight_type=QuantType.QInt8,                           #C
    per_channel=True,                                      #D
)
```

The result is 11.28 MB - 3.95× compression with only 0.10 percentage points of accuracy loss (99.11% vs 99.21%).

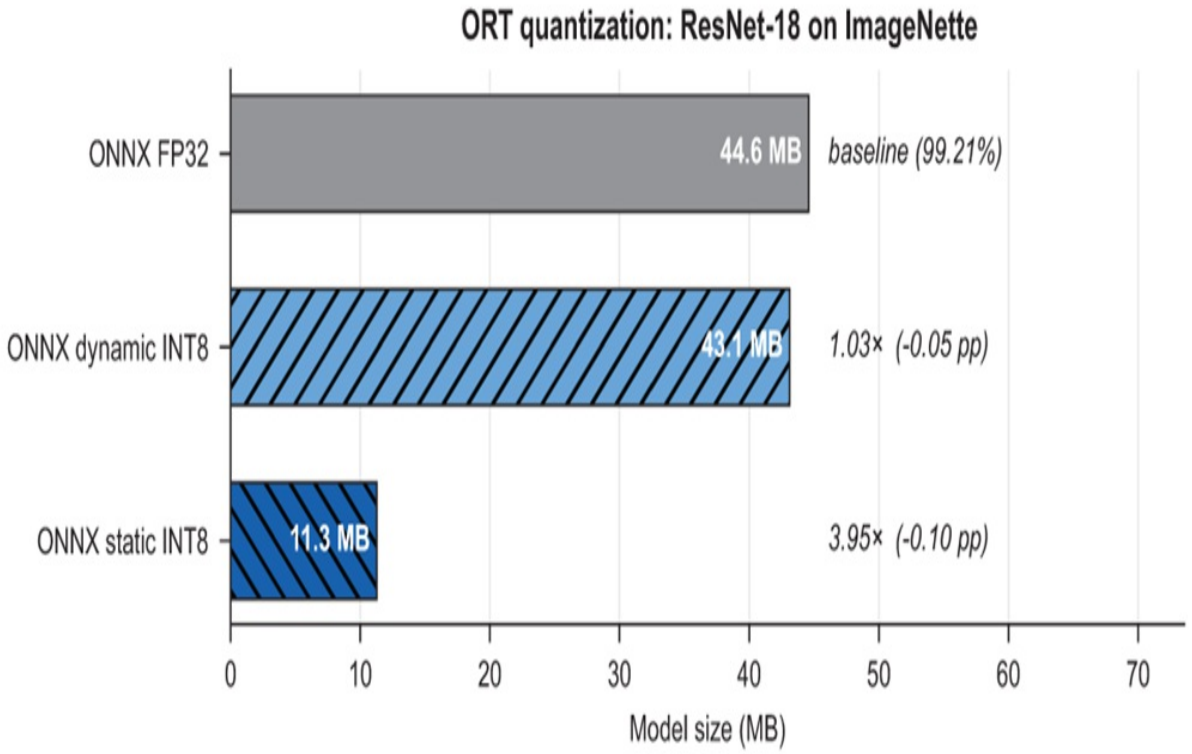
This is the payoff that section 6.2 could not deliver. TorchAO's `quantize_()` gave ResNet-18 a useless 1.03× because it only targets `nn.Linear`. ORT's `quantize_static` quantizes all 20 Conv layers, the Gemm, and their

activations.

For CNNs destined for CPU deployment, the path is clear: export FP32, let ORT quantize.

Figure 6.3 shows the three ONNX variants side by side.

Figure 6.3 ORT quantization results for ResNet-18 on ImageNette. Dynamic quantization (MatMul only) gives 1.03× - identical to TorchAO's Linear-only result. Static quantization (all operators, with calibration) gives 3.95× with 0.10 pp accuracy loss. The size gap between these two results is the cost of Conv layers staying in FP32.



For BERT, the story inverts. Dynamic quantization is the right path. ORT's `quantize_dynamic` targets all 74 MatMul nodes in the BERT graph (Q, K, V, and output projections in attention, plus FFN up and down projections across 12 encoder blocks, plus the classifier head).

The result: 105.13 MB, a 3.97× compression from the 417.85 MB FP32 baseline, with 0.34 percentage points of accuracy loss (92.09% vs 92.43%).

We also attempted static quantization on BERT. Our pipeline - TorchScript

export followed by ORT's generic `quantize_static` - produced a model that scored 50.92% accuracy, effectively random on a binary classification task. The immediate cause: ORT's generic preprocessing tool (`onnxruntime.quantization.preprocess`) failed to complete symbolic shape inference on the exported BERT graph.

Without resolved tensor shapes, calibration produced poor activation ranges, and the baked-in static scales destroyed information. The fix is specific and documented: use ORT's transformer-aware optimizer (`onnxruntime.transformers.optimizer.optimize_model` with `model_type='bert'`) instead of the generic preprocessor.

The transformer optimizer understands BERT's graph patterns - it fuses attention heads, folds layer norms, and resolves shapes correctly - producing a graph that `quantize_static` can calibrate properly. Microsoft's own BERT quantization notebooks follow this exact pipeline. We did not apply this fix here because the transformer optimizer is a runtime optimization step - it belongs alongside execution provider selection and graph optimization levels in the deployment pipeline, not in a cross-framework parity comparison.

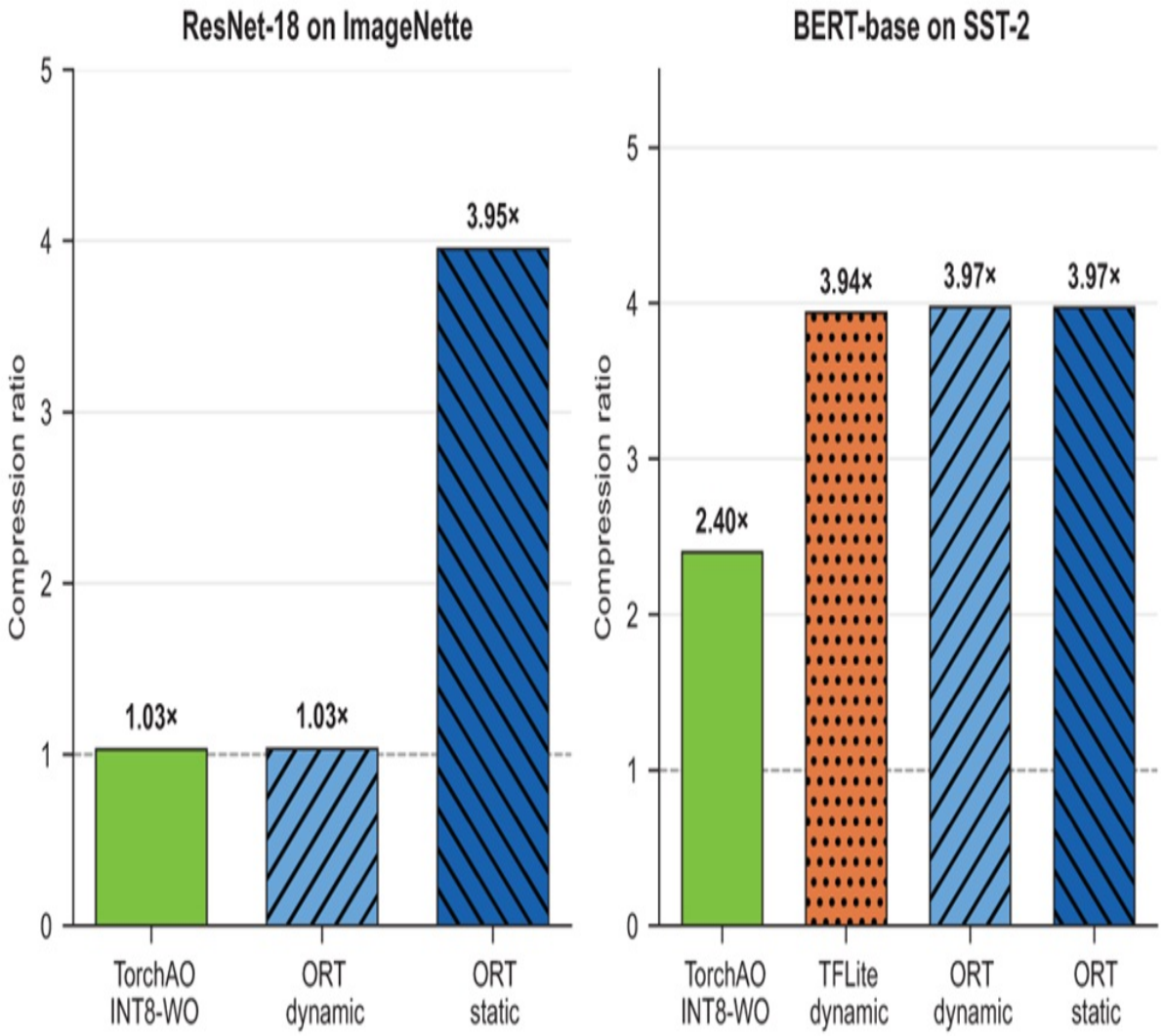
That said, even when static quantization *succeeds* on transformers through proper preprocessing, ORT's own documentation recommends dynamic quantization for transformer-based models.

The reason is architectural: transformer activations - layer norm outputs, attention scores after softmax, GELU intermediates - produce distributions that vary significantly across inputs. Static scales, computed once during calibration and frozen into the graph, cannot faithfully represent this variation.

Dynamic quantization computes activation scales per-input at inference time, adding a small compute overhead but tracking the actual distribution. For CNNs with ReLU-bounded activations, static scales work because the activation ranges are stable. For transformers, dynamic is the idiomatic path across all three frameworks we have tested: TorchAO's W8A8 dynamic (section 6.2), TFLite's dynamic range (section 6.3), and ORT's `quantize_dynamic` here.

Figure 6.4 shows the BERT-base results alongside the cross-framework comparison.

Figure 6.4 Cross-framework compression comparison. Left: ResNet-18. TorchAO and ORT dynamic both achieve 1.03× (Linear/MatMul only); ORT static gives 3.95× by quantizing all Conv layers. Right: BERT-base. TorchAO gives 2.40× (nn.Linear only), while TFLite dynamic (3.94×), ORT dynamic (3.97×), and ORT static (3.97×) all converge near 4× because they quantize all weight tensors including embeddings. The compression difference between TorchAO and the other frameworks is format scope, not quantization quality.



Choose which layers stay in FP32

Full INT8 quantization is not always the right answer. Chapter 4 established

that some layers are more sensitive to quantization than others—the first convolution (where input variance is highest) and the final classification layers (where predictions form) are common trouble spots. ORT's `quantize_static` provides a `nodes_to_exclude` parameter that keeps specified operators in FP32 while quantizing everything else. This is the ONNX-level implementation of the mixed-precision strategy from Chapter 4.

Using this knob requires knowing the node names in your ONNX graph. A quick inspection of the FP32 ResNet-18 graph reveals 20 Conv nodes and 1 Gemm node, each with a path-style name like `/layer2/layer2.0/conv1/Conv` that maps directly to the PyTorch module hierarchy. You can list them programmatically by walking `model.graph.node` in the `onnx` Python library, or open the graph visually in Netron, the standard viewer for ONNX (and TFLite and Core ML) graphs.

We tested four precision allocation strategies on ResNet-18:

The all-INT8 baseline gives 99.11% at 11.28 MB. Keeping the first convolution in FP32 costs almost nothing in size (11.30 MB) and changes nothing in accuracy (99.11%) - on this model, the input boundary is not a quantization bottleneck. Keeping the last residual block and the classifier head in FP32 jumps the model to 26.21 MB (the layer4 convolutions have 512 channels, so their FP32 weights are large) and actually *loses* 0.05 pp (99.06%) - random variation, not a real degradation. The combination of both boundary protections lands at 26.24 MB and 99.11%.

The honest assessment: ResNet-18 on ImageNette is too robust for mixed precision to matter. At 99.11% accuracy with full INT8, there is almost no accuracy to recover. The 0.10 pp gap from FP32 is within noise for 3,925 validation samples. This is exactly what a practitioner should see before reaching for mixed-precision knobs - if full INT8 works, don't complicate the deployment.

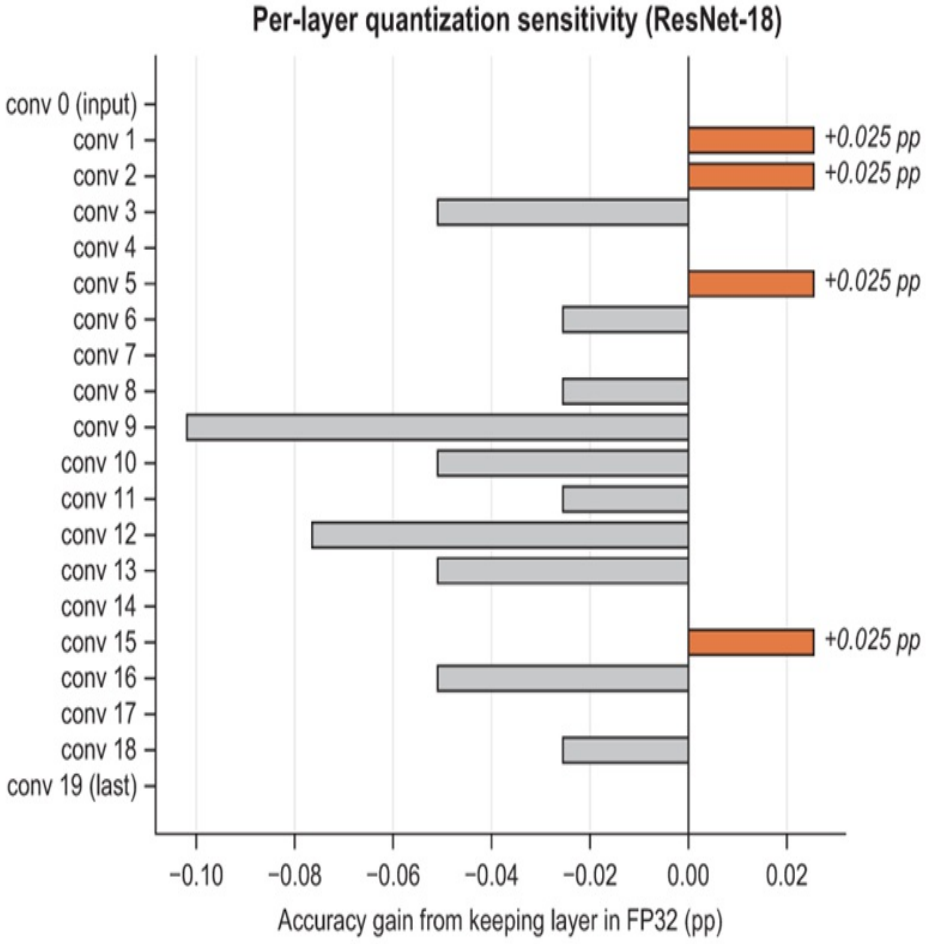
The per-layer sensitivity sweep tells a more nuanced story. For each of the 20 Conv layers, we quantize everything *except* that one layer and measure accuracy. The accuracy gain from keeping a single layer in FP32 - the layer's quantization sensitivity - ranges from +0.025 pp (four layers tied for most sensitive) to −0.10 pp (layer 9, where keeping it in FP32 slightly *hurts*

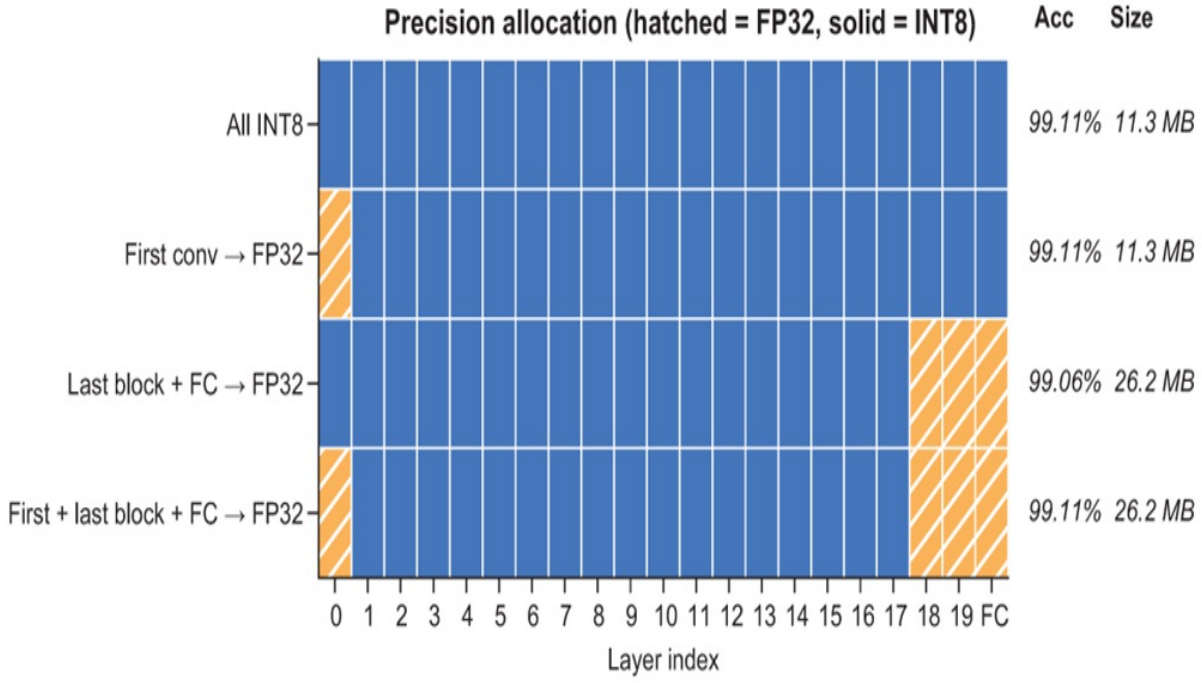
accuracy through noise). The most sensitive layers are conv 1, conv 2, conv 5, and conv 15 - the first layers in each of the four residual stages (layer1.0, layer1.0, layer2.0, layer4.0).

This pattern makes architectural sense: these are the layers where the channel count changes and the residual skip connection reshapes the feature map. On a harder task or a model with a tighter accuracy budget, these would be the layers to protect first.

Figure 6.5 shows the per-layer sensitivity and the precision allocation map.

Figure 6.5 Top: per-layer quantization sensitivity for ResNet-18. Bars extending right indicate layers that benefit from staying in FP32. The four most sensitive layers (conv 1, 2, 5, 15) are each the first convolution in a residual stage. Bottom: precision allocation map for four strategic configurations. Each row is one configuration; each column is a layer. Hatched cells are layers kept in FP32; solid cells are quantized to INT8. Accuracy and model size are annotated on the right.


Per-layer quantization sensitivity (ResNet-18)
conv 0 (input)
conv 1
conv 2
conv 3
conv 4
conv 5
conv 6
conv 7
conv 8
conv 9
conv 10
conv 11
conv 12
conv 13
conv 14
conv 15
conv 16
conv 17
conv 18
conv 19 (last)
+0.025 pp
+0.025 pp
+0.025 pp
+0.025 pp
−0.10
−0.08
−0.06
−0.04
−0.02
0.00
0.02
Accuracy gain from keeping layer in FP32 (pp)


Precision allocation (hatched = FP32, solid = INT8)
Acc
Size
All INT8
First conv → FP32
Last block + FC → FP32
First + last block + FC → FP32
99.11% 11.3 MB
99.11% 11.3 MB
99.06% 26.2 MB
99.11% 26.2 MB
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 FC
Layer index

(Full experiment code: ch6/ch6_onnx_export_path.py *--all --save-plots)*

6.5 Verify equivalence and diagnose coverage

Sections 6.2 through 6.4 built quantized artifacts through three paths: TorchAO inside PyTorch, TFLite inside TensorFlow, and ORT through ONNX. Each section reported accuracy and compression, but those metrics answer different questions than the one we need to answer now.

Accuracy tells you whether the model still works. Equivalence tells you whether the ported model produces the same outputs as the original. A model can maintain 96% accuracy in both PyTorch and ONNX Runtime while disagreeing on which specific samples it gets right — and that disagreement matters when you have downstream logic, ensembles, or regression tests that depend on deterministic behavior.

This section runs a single set of inputs through every artifact and compares the raw output tensors. The goal is to separate three sources of divergence: export error (the ONNX graph doesn't perfectly represent the PyTorch model), quantization error (INT8 arithmetic introduces rounding), and operator coverage gaps (some ops silently stay in FP32, meaning the model isn't actually running the precision you think it is).

Feed identical inputs, compare raw logits

The test is straightforward. We take 100 validation samples, run them through each variant, and compare every output logit against the PyTorch FP32 baseline. Five metrics capture different aspects of divergence: max absolute difference (the worst-case single logit), mean absolute difference (the typical error), MSE (penalizes large errors quadratically), cosine similarity (measures whether the output vector points in the same direction regardless of scale), and top-1 prediction agreement (the fraction of samples where both models pick the same class).

We compare output tensors using five metrics that capture different failure modes—max absolute difference for worst-case single-logit error, mean

absolute difference for typical drift, cosine similarity for detecting rotation of the output vector, and top-1 prediction agreement for the metric that ultimately matters in production.

Listing 6.8 Tensor-level equivalence metrics

```
def compute_equivalence_metrics(baseline, variant, labels=None):
    diff = baseline - variant
    abs_diff = np.abs(diff)

    # Per-sample cosine similarity, then average
    dot = np.sum(baseline * variant, axis=1)
    norm_b = np.linalg.norm(baseline, axis=1)
    norm_v = np.linalg.norm(variant, axis=1)
    cos_sim = dot / (norm_b * norm_v + 1e-12)

    preds_base = np.argmax(baseline, axis=1)
    preds_var = np.argmax(variant, axis=1)
    agreement = np.mean(preds_base == preds_var)

    return {
        "max_abs_diff": float(abs_diff.max()),
        "mean_abs_diff": float(abs_diff.mean()),
        "cosine_sim": float(np.mean(cos_sim)),
        "top1_agreement": float(agreement),
    }
```

NOTE

Two logit vectors can differ by a large absolute amount — say, [2.0, -1.5] versus [4.0, -3.0] — while pointing in exactly the same direction (cosine similarity 1.0). The argmax is identical in both cases. Cosine similarity captures this: it measures whether quantization rotated the output vector, not whether it scaled it.

When cosine similarity drops noticeably below 1.0, you know quantization is doing more than just rescaling — it's changing the relative ordering of logits, which is when predictions start flipping.

ResNet-18: logit divergence across four variants

We feed 100 ImageNette validation images through the PyTorch FP32 baseline and each exported variant—ONNX FP32 (no quantization, testing export fidelity alone), TorchAO INT8 weight-only, ORT Dynamic INT8, and ORT Static INT8—and compare every output logit. Table 6.9 shows the results.

Table 6.9 ResNet-18 numerical equivalence (100 ImageNette samples, vs PyTorch FP32 baseline)

Variant	Max Abs Diff	Mean Abs Diff	Cosine Sim	Top-1 Agr	Accuracy
ONNX FP32	0.00007	0.000007	1.000000	100.0%	96.00%
TorchAO INT8WO	0.146	0.021	0.999955	100.0%	96.00%
ORT Dynamic INT8	0.248	0.045	0.999813	100.0%	96.00%
ORT Static INT8	1.119	0.135	0.998236	100.0%	96.00%

The first row is the most important. ONNX FP32 — the same model exported to ONNX format with no quantization — diverges from PyTorch by at most 0.00007. That is floating-point reordering noise from the TorchScript tracer, not a modeling error. It confirms that the export step itself is essentially lossless. Any larger divergence you see in the other rows comes from quantization, not from the ONNX conversion.

The quantization rows show a clean gradient. TorchAO INT8 weight-only shifts individual logits by up to 0.146, which is what you expect when only the classification head's weights are quantized (ResNet-18's single nn.Linear layer). ORT Dynamic INT8 is slightly worse at 0.248 — it also quantizes only the Gemm operator but uses a different quantization implementation (DynamicQuantizeLinear). ORT Static INT8 quantizes every operator in the graph — all 20 Conv layers, the pooling, the residual adds — and the max logit shift rises to 1.119.

Despite this 16,000× range in max absolute difference from ONNX FP32 (0.00007) to ORT Static INT8 (1.119), every variant achieves 100% top-1

prediction agreement and identical 96% accuracy. ResNet-18's 1000-class output space provides enormous margin: a logit shift of 1.1 is noise when the winning class typically leads by 10 or more.

BERT-base: smaller output space, larger proportional error

BERT produces only two logits (positive and negative sentiment), so the same absolute quantization error occupies a much larger fraction of the output range. Where ResNet's 1000-class output space absorbs a 1-unit logit shift without noticing, BERT's 2-class output has nowhere to hide it. Table 6.10 shows the consequences.

Table 6.10 BERT-base numerical equivalence (100 SST-2 samples, vs PyTorch FP32 baseline)

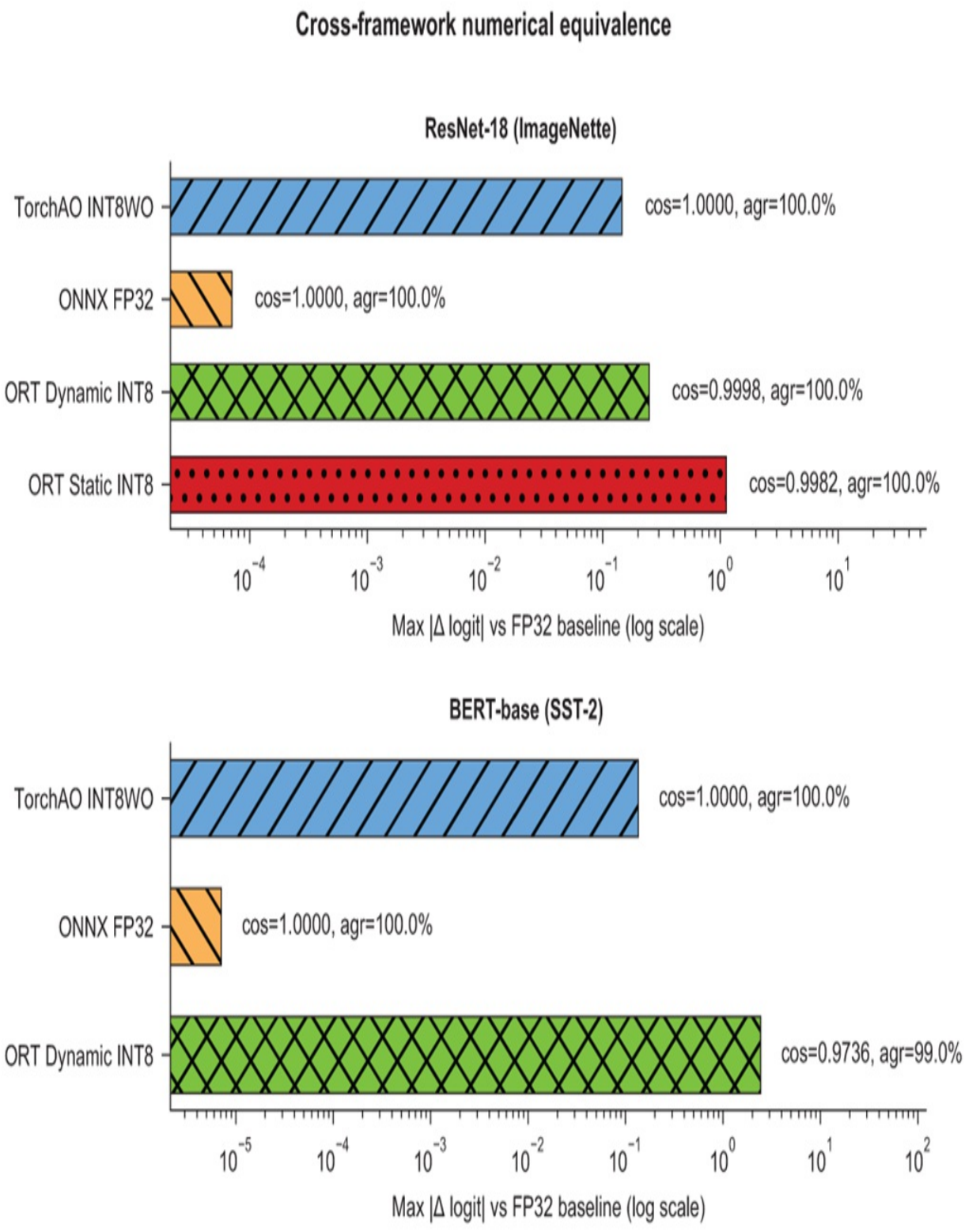
Variant	Max Abs Diff	Mean Abs Diff	Cosine Sim	Top-1 Agr	Accuracy
ONNX FP32	0.000007	0.000002	1.000000	100.0%	97.00%
TorchAO INT8WO	0.135	0.014	0.999974	100.0%	97.00%
ORT Dynamic INT8	2.428	1.448	0.973647	99.0%	96.00%

ONNX FP32 is even cleaner than vision — max |Δ| of 7e-6 confirms export fidelity. TorchAO INT8 weight-only behaves nearly identically to the vision case: max |Δ| of 0.135 with perfect prediction agreement.

ORT Dynamic INT8 is where the story gets interesting. The max logit shift is 2.428 — more than double ResNet's ORT Static (1.119) — even though BERT Dynamic quantizes only 28% of operators compared to ResNet Static's 100%. The mean absolute difference is 1.448, and cosine similarity drops to 0.9736. One sample out of 100 flips its prediction, dropping accuracy from 97% to 96%.

Figure 6.6 puts all variants on the same log-scale axis, making the magnitude of divergence across frameworks and models visually comparable.

Figure 6.6 Cross-framework numerical equivalence for ResNet-18 (top) and BERT-base (bottom). Bars show max |Δ logit| vs the PyTorch FP32 baseline on a log scale. ONNX FP32 export introduces negligible divergence (~10^{-5}). Quantization shifts logits by 10^{-1} to 10^{0}, but prediction agreement remains above 99% for all variants. The largest divergence is ORT Dynamic INT8 on BERT — cosine similarity 0.9736 — where the narrow 2-class output space amplifies quantization error.

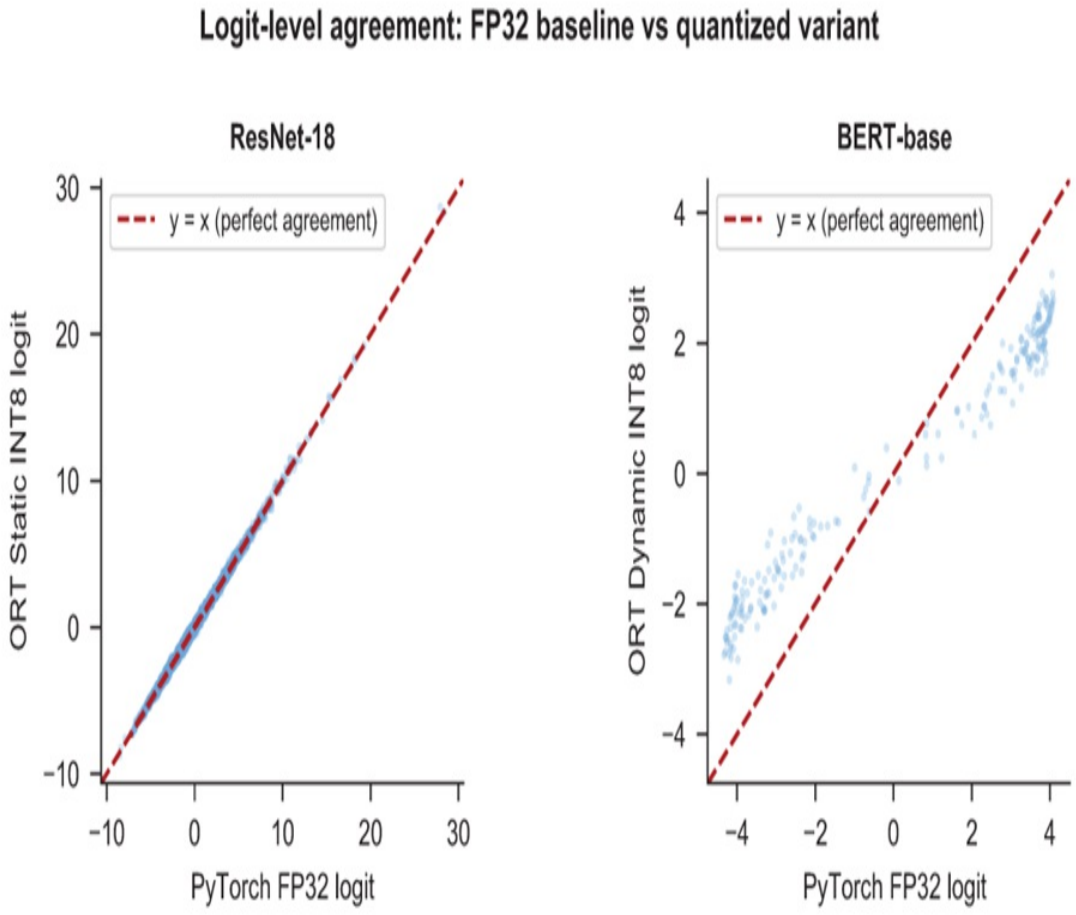


The aggregate metrics compress each variant into a single number. To see where individual predictions actually drift, we plot every logit from every

sample: the PyTorch FP32 value on the x-axis, the quantized variant's value on the y-axis. Points on the diagonal mean perfect agreement; points off it are where quantization moved a logit.

Figure 6.7 shows this for the two most informative comparisons.

Figure 6.7 Logit-level agreement: FP32 baseline vs quantized variant. Left: ResNet-18 vs ORT Static INT8 (1000 logits × 100 samples). Points cluster tightly along the y = x identity line across the full [-10, +30] range despite 100% operator coverage. Right: BERT-base vs ORT Dynamic INT8 (2 logits × 100 samples). Visible scatter away from the diagonal, especially for negative logits around -4, with several points drifting by more than 1 unit. The narrow output range makes the same absolute error proportionally larger.



The visual confirms the arithmetic. ResNet's 1000-dimensional output absorbs quantization noise without visible deviation from the identity line. BERT's 2-dimensional output shows individual points drifting visibly off the diagonal — some by more than 2 logit units. The cluster of points near (-4,

-4) on the BERT panel shows where negative-class logits shift enough to approach the decision boundary, which is why one prediction actually flips.

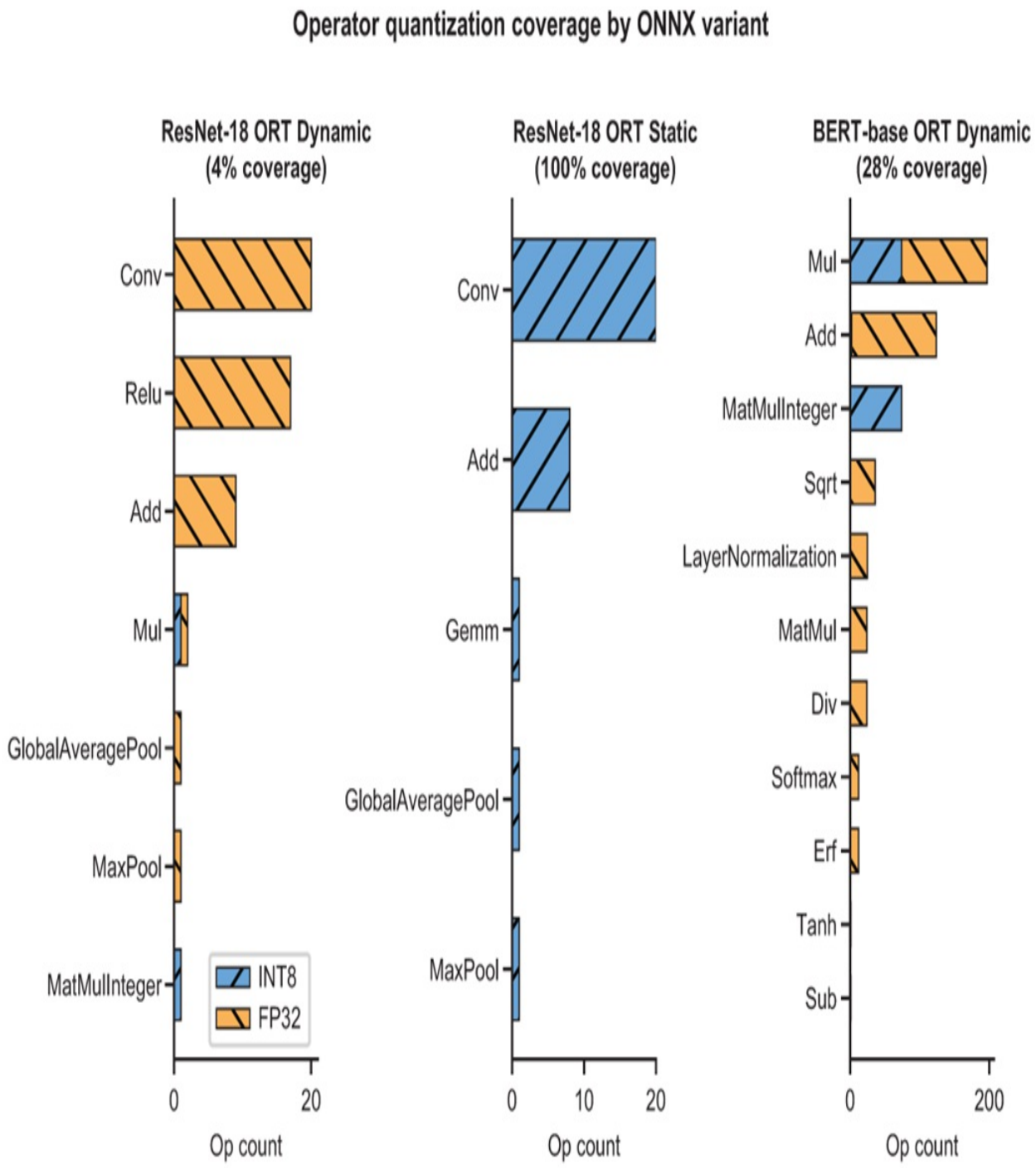
Diagnose coverage: why quantization doesn't mean faster

The equivalence test answers "are the outputs the same?" The operator coverage analysis answers a different question: "how much of the graph actually runs in INT8?" Section 6.2 showed that eager-mode quantization doesn't speed up inference — latency improvement requires a runtime that can execute integer arithmetic natively. But even with the right runtime, speedup only happens for operators that actually received quantization. If 96% of your compute stays in FP32, you get roughly 0% speedup.

Our companion script on GitHub inspects each ONNX graph and reports, for every operator type, how many instances execute in INT8 versus FP32. For QDQ-format graphs (like ORT Static), an operator is "quantized" if its inputs come from DequantizeLinear nodes — the runtime fuses the QDQ sandwich into a native INT8 kernel. For operator-replacement format (like ORT Dynamic), quantized operators appear as distinct nodes like MatMulInteger or DynamicQuantizeLinear.

Figure 6.7 shows the operator coverage for three quantized variants.

Figure 6.8 Operator quantization coverage by ONNX variant. Left: ResNet-18 ORT Dynamic — 4% coverage, only 1 MatMulInteger node (the classification head's Gemm) out of 51 compute ops. All 20 Conv and 17 Relu stay in FP32. Center: ResNet-18 ORT Static — 100% coverage, all 20 Conv, 8 Add, and the Gemm wrapped in QDQ pairs. Right: BERT-base ORT Dynamic — 28% coverage, 74 MatMulInteger nodes (the linear projections) quantized, but LayerNormalization, Softmax, Div, and Sqrt remain in FP32.



The coverage numbers reveal why compression and speedup don't always track together. ResNet-18 Dynamic's 4% coverage explains why section 6.4 reported 1.03× compression for dynamic quantization—identical to TorchAO's Linear-only result. Dynamic quantization targets MatMul and Gemm operators by default. ResNet-18's compute is 20 Conv layers, not

matrix multiplications. Only the single classification head (Gemm) gets quantized, leaving 96% of compute untouched.

ResNet-18 Static tells the opposite story: 100% coverage because QDQ-format quantization wraps every operator — Conv, Add, MaxPool, GlobalAveragePool — in QuantizeLinear/DequantizeLinear pairs. The graph goes from 48 compute ops to 31 (the runtime fuses the QDQ nodes into their consumers), with 108 QDQ nodes orchestrating the data type transitions. This is why static quantization achieved 3.95× compression with 0.10 pp accuracy loss in section 6.4.

The most surprising result is BERT Dynamic's 28% coverage, which looks low but is deceptive. The 74 MatMulInteger nodes correspond to the Q, K, V, output, and FFN projections across 12 encoder layers — the compute-heavy linear algebra. The remaining 380 FP32 ops are mostly element-wise: 122 Adds, 36 Sqrts, 24 Divs, 25 LayerNormalizations, 12 Softmaxes. These are cheap individually but numerous. By parameter count and FLOP share, the 74 quantized MatMul operations dominate BERT's compute budget, which is why section 6.4 reported 3.60× compression despite the low op-count coverage percentage.

When coverage lies to you

The coverage percentage alone is misleading. BERT Dynamic's 28% op coverage conceals the fact that those 74 MatMulInteger nodes account for the vast majority of BERT's floating-point operations. Conversely, a graph could report 80% coverage while leaving the single most expensive operator in FP32, producing negligible speedup.

The right diagnostic is to cross-reference coverage with a profiling tool — ORT's built-in profiler (`sess_options.enable_profiling = True`) or the ONNX graph's FLOPs — to verify that quantized operators account for the bulk of wall-clock time, not just the bulk of node count.

A quick-reference troubleshooting map

When cross-framework validation fails or quantized models don't deliver

expected speedups, the cause usually falls into one of these categories:

Logits diverge but accuracy holds. This is normal for INT8 quantization. If cosine similarity stays above 0.99 and top-1 agreement stays above 99%, the quantization is working correctly. The absolute logit differences are rounding noise amplified through the network.

Logits diverge and accuracy drops. Check operator coverage first — if most ops stayed in FP32, the accuracy drop comes from the few that were quantized, and they may be sensitive layers. Review the per-layer sensitivity analysis: keep sensitive layers in FP32 and re-run.

ONNX FP32 diverges from PyTorch by more than 1e-3. This indicates an export problem, not a quantization problem. Common causes: the dynamo exporter failed silently and fell back to TorchScript with different graph rewriting, custom operators weren't registered, or dynamic axes weren't specified and the graph was specialized to the tracing input shape.

Model quantized but no speedup. Inspect operator coverage. If compute-heavy ops (Conv, MatMul, Gemm) show ✗ in the INT8 column, the runtime is executing those in FP32. For dynamic quantization, this is by design — it targets MatMul/Gemm only. For static quantization, missing coverage usually means the preprocessing step failed to insert QDQ nodes around certain ops, typically because of unusual graph patterns or unsupported op types.

ORT session creation fails with "Not Implemented." The runtime lacks a kernel for the quantized operator on your hardware. This is common with ConvInteger on CPU-only builds. The fix is to either use QDQ format (which lets the runtime decide whether to fuse into INT8 or fall back gracefully) or restrict `op_types_to_quantize` to operators with known kernel support.

(Full experiment code: ch6/ch6_verify_equivalence.py --all --save-plots)

Porting a quantized model across frameworks is an exercise in controlling divergence — knowing where default semantics, operator fusion, and calibration implementations disagree, then validating that none of those disagreements flip predictions. The models tested here — ResNet-18,

MobileNetV2, BERT-base, TinyLlama — are small enough that standard post-training quantization handles them without special treatment. At the scale of billions of parameters, activation outliers grow large enough to break that assumption, and the techniques covered so far cannot recover the loss. Chapter 7 introduces the specialized methods — LLM.int8(), GPTQ, AWQ — that the LLM community developed in response.

6.6 Summary

- Quantization workflows diverge across frameworks in default semantics, operator fusion rules, and calibration implementations. Parity requires tensor-level validation, not just accuracy checks.
- TorchAO's quantize_() targets nn.Linear only—compressing BERT-base by 2.40× and TinyLlama by 3.38×, but leaving Conv2d-dominated ResNet-18 at 1.03×.
- TFLite's converter handles quantization, optimization, and serialization in one pass. Dynamic range achieves 3.68× on MobileNetV2 with no calibration data.
- ONNX export is essentially lossless (max logit divergence ~10^{-5}). Larger divergence in quantized variants comes from quantization arithmetic, not the export.
- ORT dynamic quantization targets MatMul/Gemm, effective for transformers (3.97× on BERT) but not CNNs (1.03× on ResNet-18). ORT static with QDQ format quantizes all operators (3.95× on ResNet-18).
- Accuracy preservation does not guarantee numerical equivalence. BERT under ORT Dynamic showed one prediction flip in 100 samples despite maintaining 96% overall accuracy.
- Operator coverage determines speedup—low op-count coverage can still be effective if those operators dominate compute.
- When validation fails, compare ONNX FP32 against the PyTorch baseline first to isolate export error from quantization error.

7 Quantizing Large Language Models in Practice

This chapter covers

- Why LLMs need a new quantization toolkit: activation outliers and KV-cache pressure
- LLM.int8(): routing outlier dimensions to FP16
- GPTQ: Hessian-guided error compensation for 4-bit weights
- AWQ: activation-aware scaling to protect salient channels
- TurboQuant for the KV cache, and choosing a method for your deployment

Quantization so far has meant squeezing a well-behaved tensor into fewer bits. Large language models break that assumption. A handful of hidden dimensions inside a 7B or 70B model carry magnitudes hundreds of times larger than their neighbors, and those dimensions are load-bearing — clip them and the model's perplexity collapses by an order of magnitude.

Any practitioner shipping an LLM runs into this the first time they try to serve one on commodity GPUs. Memory, concurrent users, and tokens-per-second all hinge on how you handle these outliers, and on a second bottleneck that only appears at serving time: the KV cache, which at long context can equal the model itself in size.

Four production methods — LLM.int8(), GPTQ, AWQ, and TurboQuant — each solve a piece of the puzzle. We measure where the standard toolkit breaks, build each method up, and finish with a deployment decision.

7.1 Handle outliers and key-value cache considerations

At BERT scale, the worst activations sit roughly 5–6× above the channel

median, and percentile clipping or MSE-optimal calibration handles them. At ~6.7 billion parameters, three things change.

First, outlier magnitudes jump to 100× the median and beyond. *Second*, the high-magnitude channels stop being layer-specific accidents and become the same small set of hidden dimensions firing across every layer — a coordinated structure we can exploit. *Third*, those same channels stop being disposable; clipping them collapses the model. Per-tensor INT8 fails everywhere from 6.7B on up, and the rest of this chapter is the toolkit for getting accuracy back.

We measure each effect in turn, starting with coordination, which is the most visually striking and the property every later method relies on.

Outlier channels become coordinated

In BERT and OPT-125M, the highest-magnitude channels vary across layers —different layers have different hot dimensions. To measure whether this changes at scale, we hook into every fc1 layer's input activations and record per-channel maxima over WikiText-2 calibration data.

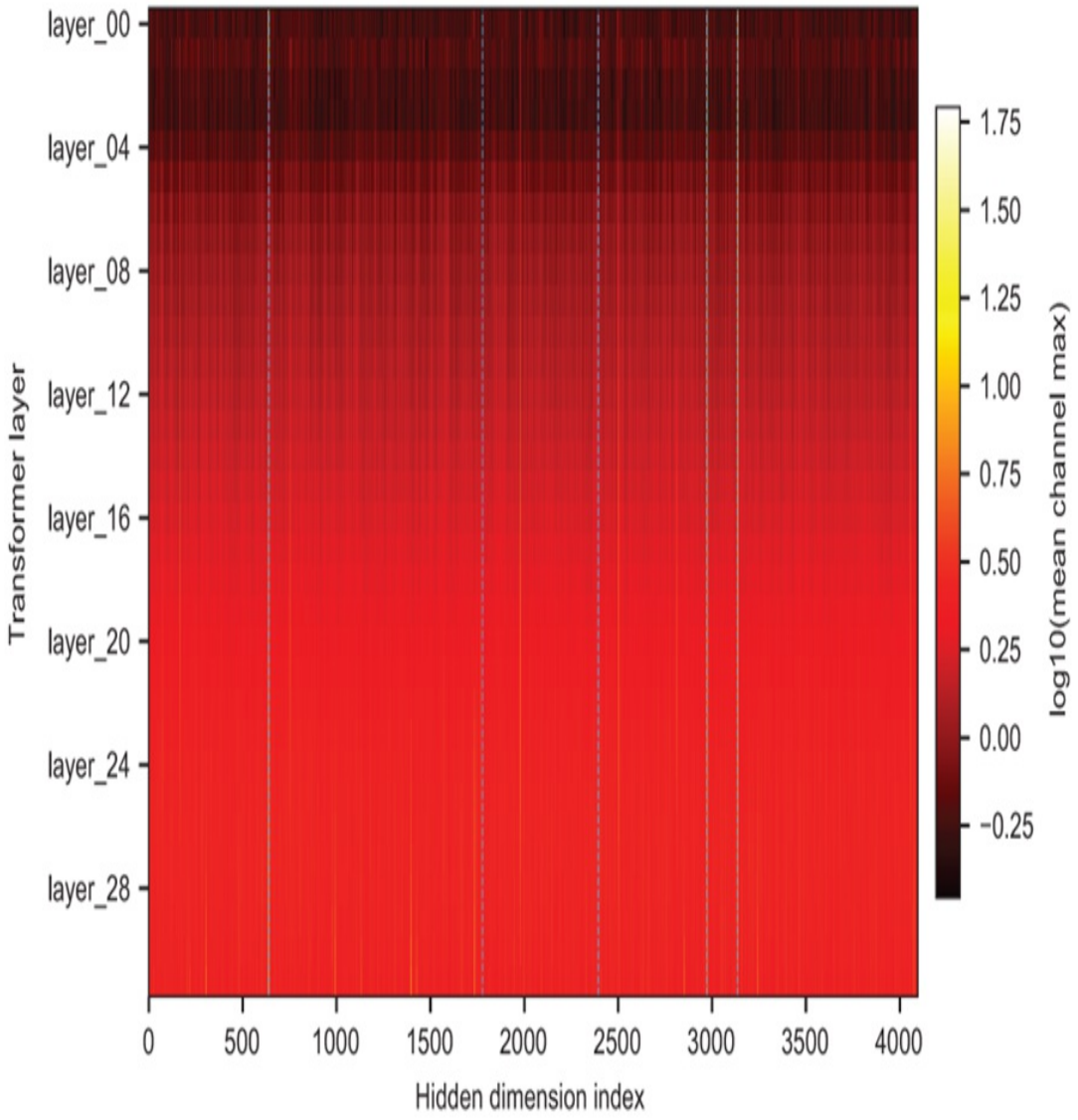
Listing 7.1 Profiling per-channel activation magnitudes

```
layer_channel_maxes = defaultdict(list)

def make_hook(name):
    def hook_fn(module, input, output):
        x = input[0].detach().float()
        flat = x.reshape(-1, x.shape[-1])              #A
        channel_max = flat.abs().amax(dim=0)            #B
        layer_channel_maxes[name].append(channel_max.cpu())
    return hook_fn

for layer_idx in range(num_layers):                     #C
    layer = model.model.decoder.layers[layer_idx]
    layer.fc1.register_forward_hook(
        make_hook(f"layer_{layer_idx:02d}")
    )
```

Running this on OPT-125M and OPT-6.7B (Colab T4 GPU) produces the

heatmap in Figure 7.1: each row is a transformer layer, each column one of 4,096 hidden dimensions, brightness the log-scale mean channel maximum. The T4 here is only running the profiling sweep — the heatmap is a property of the trained weights, not the GPU it ran on.

Figure 7.1 Per-channel activation magnitude heatmap for OPT-6.7B (32 layers × 4,096 hidden dimensions, log scale). Bright vertical stripes mark the same hidden dimensions spiking across all layers. Dashed lines mark the five universal outlier channels—dimensions 639, 1778, 2394, 2972, and 3136—that appear in the top 1% of every layer.

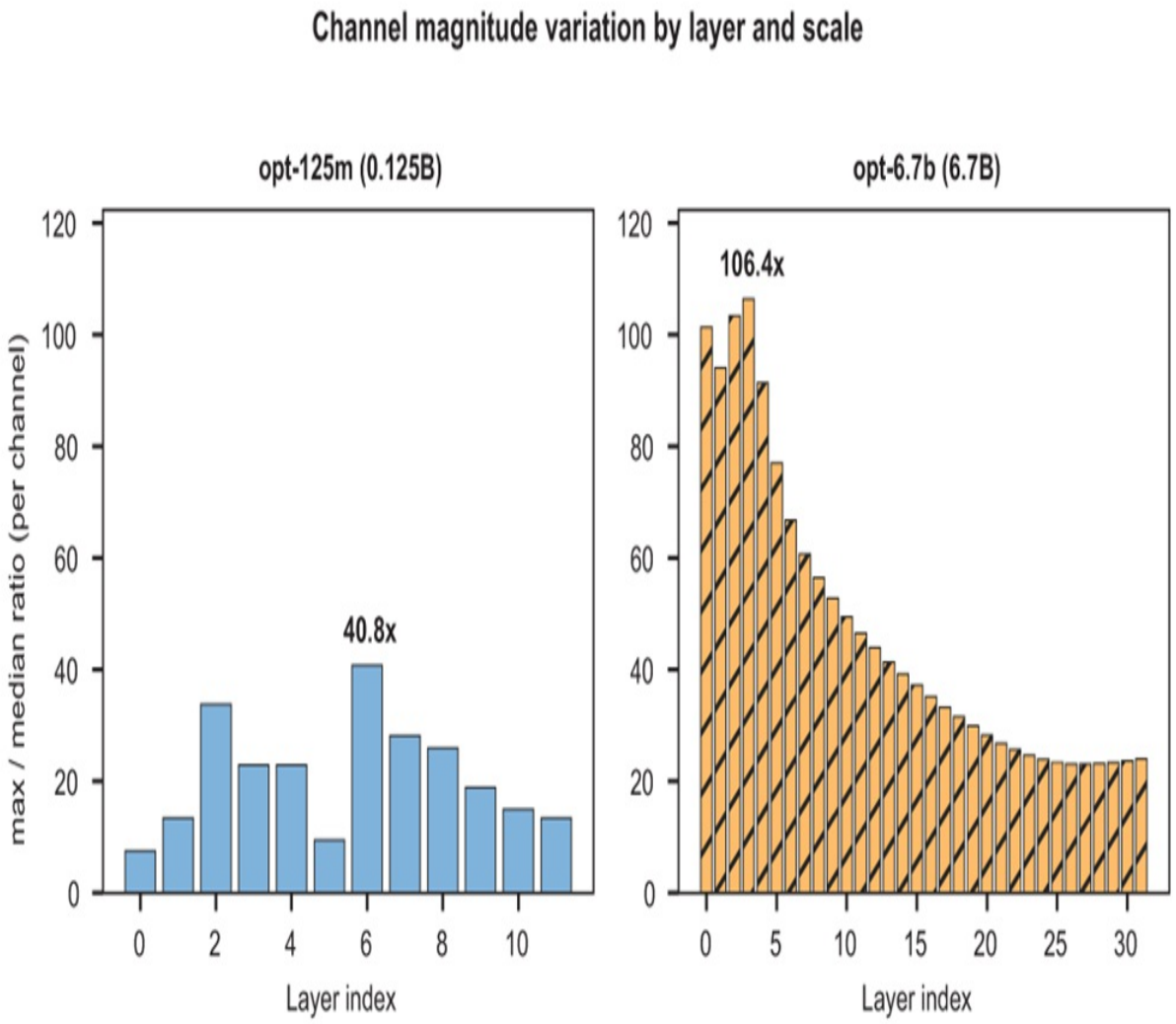


Five hidden dimensions appear in the top 1% of every single layer. The Jaccard similarity of top channels between consecutive layers is 0.70—

compared to 0.54 for OPT-125M, where no channels are universally dominant. This coordination is the qualitative change at scale: the network has learned to use specific hidden dimensions as high-magnitude information highways across the entire model. It is also what makes methods like LLM.int8() feasible—outlier dimensions can be identified once and routed globally.

The coordination is visible. The magnitude is the second change. Figure 7.2 shows the per-layer max/median ratio for both models.

Figure 7.2 Channel magnitude variation across model scales. Left: OPT-125M shows sporadic spikes with no consistent pattern across layers. Right: OPT-6.7B shows sustained high ratios (max/median above 23× everywhere, 106× peak) that decrease smoothly from early to late layers. Shared Y-axis makes the severity contrast visible. Numbers above bars indicate the peak ratio per model.



MSE-optimal calibration handles outlier ratios up to about 10×. At 106×, it runs out of room: a per-tensor INT8 scale set by a magnitude-60 channel gives a step size of 60/127 ≈ 0.47, leaving the 99% of channels with magnitudes below 3 only ~6 usable INT8 levels.

Per-channel quantization — one scale per output channel — recovers most of the resolution. But the outlier dimensions still account for 82% of residual error in early layers, as Figure 7.3 shows.

Outliers become unclippable

Percentile clipping is the standard escape valve when outliers blow up the quantization range: sacrifice the tail to gain resolution for the majority. At

6.7B, this fails.Dettmers et al. (2022) showed that zeroing the emergent outlier channels degrades perplexity by 600–1,000%. The model has concentrated critical function into a few dimensions that cannot be clipped or rounded away. The rest of the chapter is about preserving those dimensions rather than suppressing them.

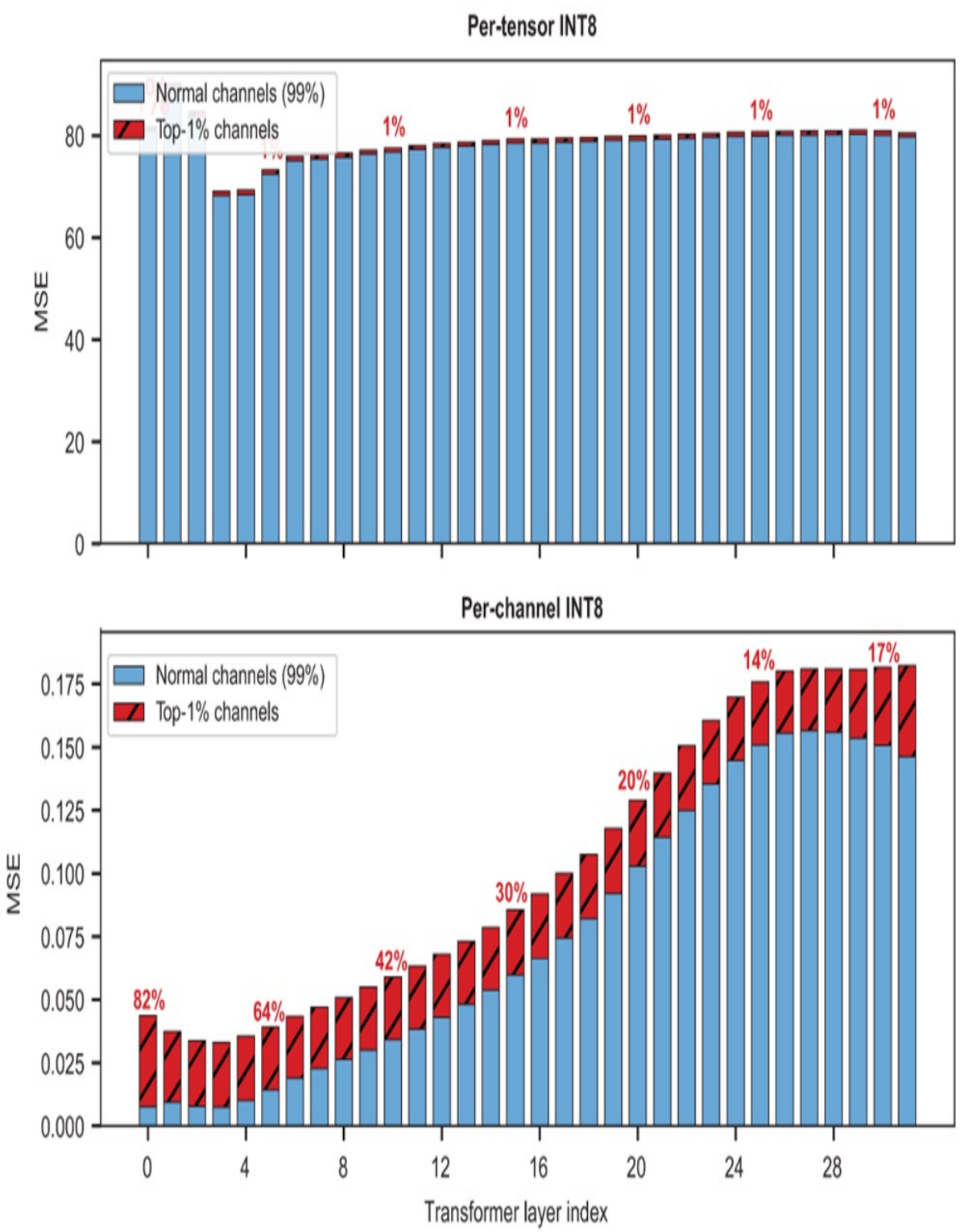
To see exactly where the damage lands and why scale-poisoning is the right name for it, Listing 7.2 decomposes INT8 quantization error into the contribution from the top-1% of channels versus the rest.

Listing 7.2 Quantization error decomposition

```
def quantize_per_tensor_int8(tensor):
    scale = torch.clamp(tensor.abs().max() / 127, min=1e-8)
    q = torch.round(tensor / scale).clamp(-128, 127)
    return q * scale                                    #A

def quantize_per_channel_int8(tensor):
    scale = torch.clamp(tensor.abs().amax(dim=0) / 127, min=1e-8)
    q = torch.round(tensor / scale).clamp(-128, 127)
    return q * scale                                    #B

flat = x.reshape(-1, x.shape[-1])
mse_pt = ((flat - quantize_per_tensor_int8(flat)) ** 2).mean(dim=
mse_pc = ((flat - quantize_per_channel_int8(flat)) ** 2).mean(dim
```

Figure 7.3 shows the result for OPT-6.7B.

Figure 7.3 Where quantization error comes from in OPT-6.7B. Top: under per-tensor INT8, the top-1% channels (40 of 4,096) contribute only ~1% of MSE—they set the scale and quantize cleanly. The 99% of normal channels absorb the error. Bottom: per-channel INT8 reduces total MSE by 1,872× (layer 0) to 442× (layer 31), but outlier dims still account for 82% of residual error in early layers—motivating decomposition into separate precision paths.


Where quantization error comes from (opt-6.7b)
Per-tensor INT8
Normal channels (99%)
Top-1% channels
MSE
0
20
40
60
80
1%
1%
1%
1%
1%
Per-channel INT8
Normal channels (99%)
Top-1% channels
MSE
0.000
0.025
0.050
0.075
0.100
0.125
0.150
0.175
82%
64%
42%
30%
20%
14%
17%
0
4
8
12
16
20
24
28
Transformer layer index

Per-channel quantization reduces error by three orders of magnitude, but the residual concentration in outlier dimensions confirms that uniform-precision approaches are insufficient at this scale.

KV cache: the serving multiplier

Compressing the weights shrinks the model, but at serving time a second tensor competes for the same GPU memory: the key-value cache. For every token generated, every layer appends its keys and values; over thousands of tokens and dozens of layers, the cache can rival or exceed the model in size. Table 7.1 translates that into concurrent users on an 80 GB A100.

Table 7.1 Concurrent users on an 80GB A100 serving Llama-3.1-8B (INT4 weights, ~4 GB model)

Context length	FP16 KV per user	Max users	INT4 KV per user	Max users	Multiplier
2,048	0.25 GB	296	0.06 GB	1,184	4.0×
8,192	1.00 GB	74	0.25 GB	296	4.0×
32,768	4.00 GB	18	1.00 GB	74	4.1×
131,072	16.00 GB	4	4.00 GB	18	4.5×

Computed from architecture parameters: 74 GB available for KV (80 GB minus ~4 GB INT4 weights, ~2 GB overhead). At 128K context, a single user's FP16 KV cache equals the model—INT4 KV is the difference between serving 4 and 18 concurrent users. KV cache quantization runs alongside the weight and activation methods in Sections 7.2–7.4 and compounds their memory savings.

NOTE

The mental anchor for LLM quantization: The scale of the problem changes qualitatively at LLM scale: outlier magnitudes jump from 6× to 106× the channel median (far beyond what calibration can resolve), they concentrate in a handful of dimensions that persist across all layers (Jaccard 0.70, 5

universal dims), and they cannot be clipped away — removing them collapses perplexity by 600–1,000%. Every method in this chapter is an answer to one or both of those problems, combined with KV cache quantization for serving capacity.

Reproducing these results: The companion script ch7/ch7_outlier_profiling.py generates all experiments and figures from this section.

```
# Full pipeline on Colab T4
python ch7_outlier_profiling.py --mode all --save-plots

# CPU-only (runs opt-125m, skips 6.7b)
python ch7_outlier_profiling.py --mode all --save-plots \
    --model opt-125m --emergence-models opt-125m
```

7.2 Apply outlier-aware weight paths

We now have a puzzle to solve: per-channel INT8 reduces quantization error by three orders of magnitude, yet outlier dimensions still concentrate 82% of the residual error in early layers.

The coordination property - the same 4-7 dimensions spiking in every layer, Jaccard similarity 0.70 - points toward the solution. If the outliers are systematic and sparse, we don't need to find a single scale that compromises between them and everything else. We can route them through a separate compute path entirely.

This is the core idea behind LLM.int8() (Dettmers et al., 2022): split each matrix multiplication into two paths based on activation magnitude, compute the outlier dimensions in full precision and the remaining 99.9% in INT8, then sum the results. The method requires no calibration data, no retraining, and no model modification - it operates dynamically on each forward pass.

Build the decomposition from scratch

The algorithm has four steps: detect outlier feature dimensions in the activation tensor, split both the activations and weights into outlier and

normal columns, compute each path at its own precision, and sum the outputs.

Listing 7.3 implements the INT8 quantization used for the normal path, and Listing 7.4 implements the full decomposition.

Listing 7.3 Per-channel absmax INT8 quantization

```
def quantize_absmax_int8(tensor):
    scales = tensor.float().abs().amax(dim=1, keepdim=True) / 127
    scales = torch.clamp(scales, min=1e-8)
    q_int8 = torch.round(tensor.float() / scales).clamp(-128, 127
    return q_int8.to(torch.int8), scales
```

Each of the 4,096 output channels gets its own scale, giving 127 usable INT8 levels per channel.

By itself, per-channel weight quantization introduces negligible error - weight distributions at 6.7B are well-behaved, with no extreme outliers.

The damage at this scale comes from *activation* quantization, not weights.

Listing 7.4 Mixed-precision decomposition (the LLM.int8() algorithm)

```
def mixed_precision_matmul(X, W, threshold=6.0):
    # Step 1: Identify outlier feature dimensions
    outlier_mask = X.abs().max(dim=0).values > threshold
    outlier_dims = outlier_mask.nonzero(as_tuple=True)[0]
    normal_dims = (~outlier_mask).nonzero(as_tuple=True)[0]

    # Step 2: Split into two paths
    X_outlier = X[:, outlier_dims].float()
    W_outlier = W[:, outlier_dims].float()
    X_normal = X[:, normal_dims].float()
    W_normal = W[:, normal_dims]

    # Step 3a: Full-precision matmul for outlier dimensions
    out_outlier = X_outlier @ W_outlier.T

    # Step 3b: INT8 matmul for normal dimensions
    W_q, W_s = quantize_absmax_int8(W_normal)
    X_scales = X_normal.abs().amax(dim=1, keepdim=True) / 127.0
```

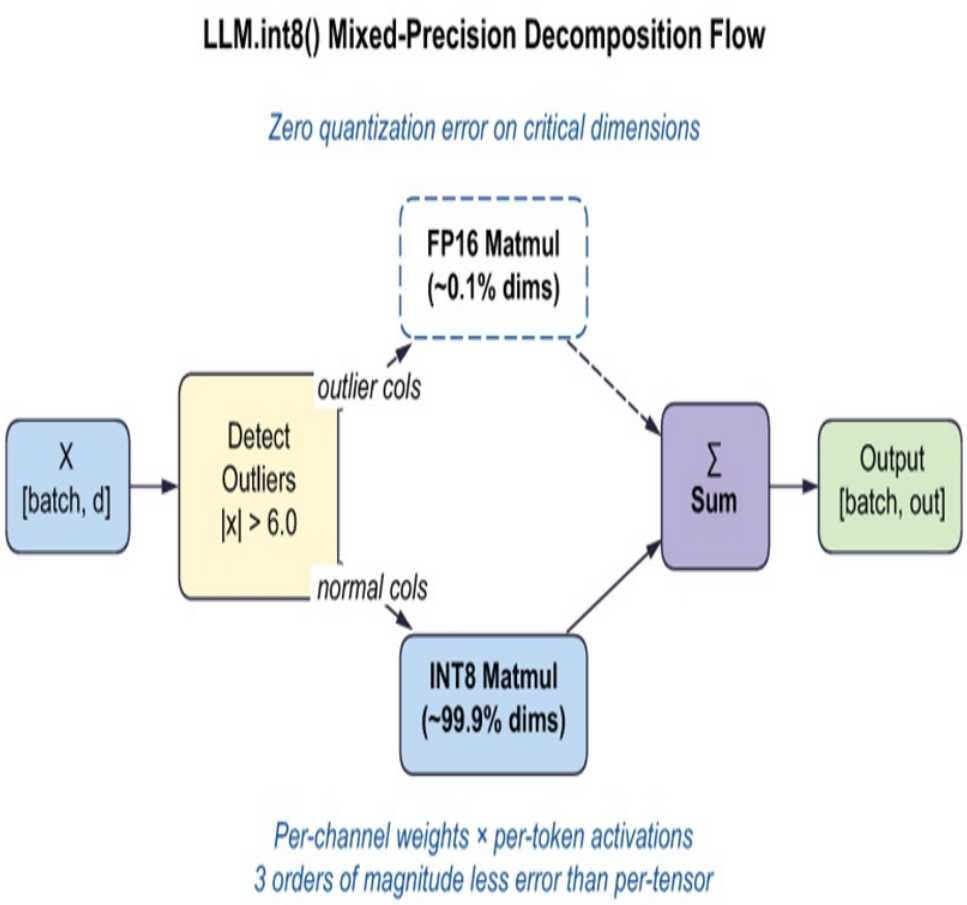
```
    X_q = torch.round(X_normal / X_scales).clamp(-128, 127)
    out_normal = (X_q.float() @ W_q.float().T) * (X_scales * W_s.

    # Step 4: Sum the two paths
    return out_outlier + out_normal
```

A few design choices deserve attention. Outlier detection runs on activations, not weights: the 100×-style magnitudes are an activation phenomenon, while weight distributions stay well-behaved.

On the normal path, activation quantization is per-token (row-wise) rather than per-tensor, because different tokens have different magnitude profiles and a single high-magnitude token would corrupt the scale for the entire batch. The split itself operates on the feature dimension (columns of both X and W), keeping the two matmul outputs the same shape so they can be summed directly.

Figure 7.4 shows the data flow.

Figure 7.4 LLM.int8() mixed-precision decomposition flow. Input activations are scanned for outlier dimensions (|x| > 6.0). The ~0.1% of dimensions that exceed the threshold are routed to a full-precision matmul (dashed path); the remaining ~99.9% are quantized to INT8. Summing the two outputs recovers the full result with zero quantization error on the critical dimensions and per-channel × per-token precision on everything else.



To confirm that the decomposition works as advertised, we apply Listing 7.4 to every fc1 (FFN up-projection) layer in OPT-6.7B and compare the reconstruction error against the FP16 ground truth.

For each layer, we capture activations from WikiText-2 calibration data, compute the FP16 matmul as reference, then measure the MSE of both naive INT8 (no decomposition) and the mixed-precision path. Table 7.2 shows selected layers.

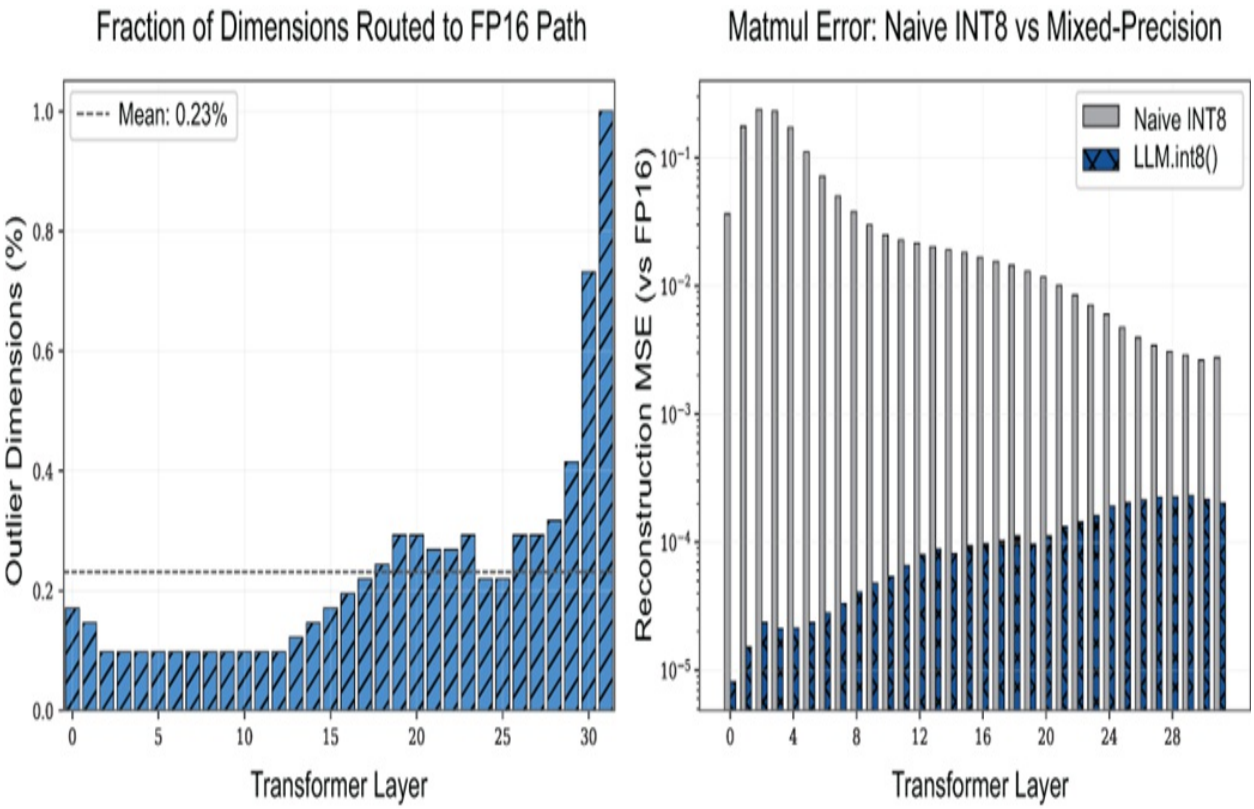
Table 7.2 Decomposition anatomy for OPT-6.7B fc1 layers (threshold 6.0)

Layer	Outlier dims	Outlier %	INT8 MSE	Mixed MSE	Ratio
0	7	0.17%	0.0366	0.000008	4,525×
1	6	0.15%	0.1765	0.000015	11,642×
5	4	0.10%	0.1108	0.000024	4,701×

15	7	0.17%	0.0182	0.000093	195×
20	12	0.29%	0.0117	0.000111	105×
25	9	0.22%	0.0047	0.000203	23×
31	41	1.00%	0.0028	0.000202	14×
Avg	8.3	0.23%			1,827×

Figure 7.5 shows the full 32-layer picture.

Figure 7.5 Decomposition anatomy for OPT-6.7B. Left: fraction of dimensions routed to the FP16 path per layer (mean 0.23%, dashed line). The overhead increases from ~0.1% in early layers to 1.0% in the last layer, but stays below 1% everywhere except layer 31. Right: reconstruction MSE vs FP16 on a log scale. Filled bars show naive INT8 error; hatched bars show the mixed-precision error. The gap ranges from 4 orders of magnitude in early layers to just over 1 order in late layers.



The data reveals a consistent picture. The FP16 overhead is negligible: an average of 0.23% of dimensions are routed to the full-precision path, meaning 99.77% of the compute stays in INT8. This is not a fallback or a

compromise—it is a surgical excision of the dimensions that would poison the scale for everything else.

There is also a clear gradient across the model. Early layers (0–5) show error reduction ratios of 4,500–11,600× because their few outlier dims (4–7) sit at magnitudes of 60–106×. Late layers (28–31) show ratios of only 12–14× because they have more outlier dims (30–41) with moderate magnitudes.

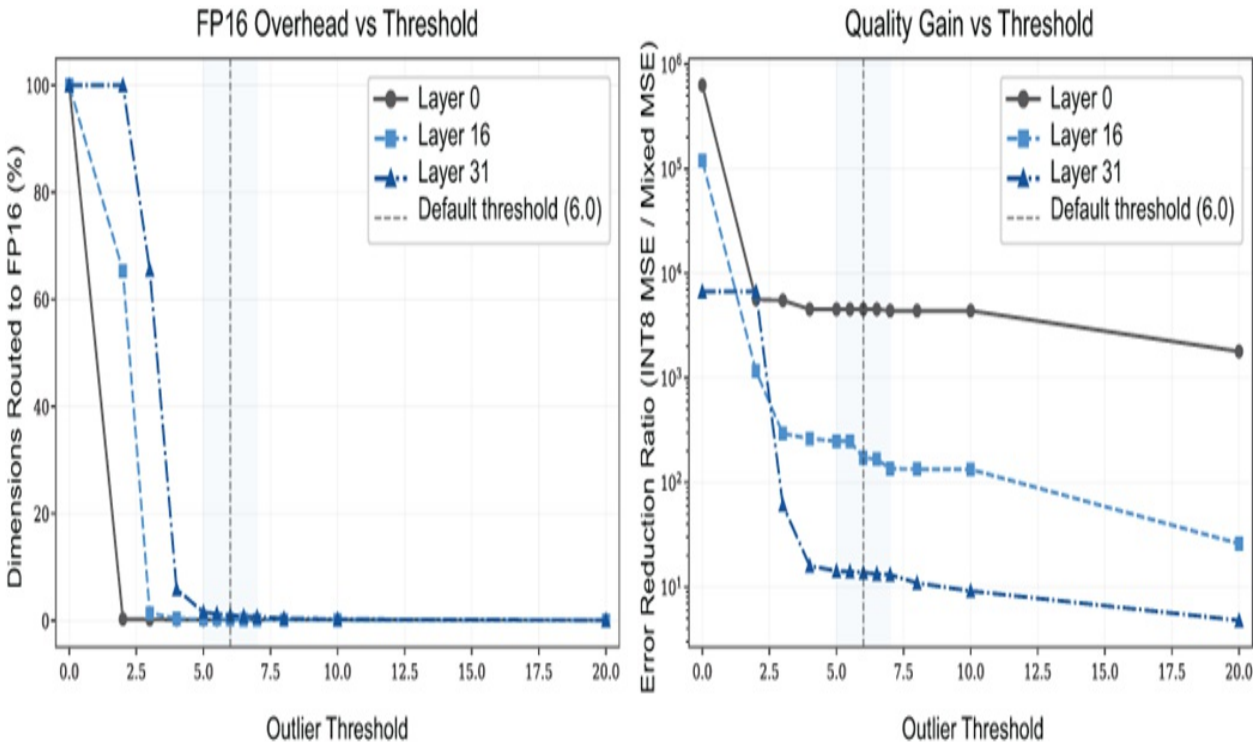
The decomposition helps everywhere, but the early layers are the high-leverage targets.

The structural consistency goes further. The four dimensions appearing in more than 80% of layers — 1778, 2394, 2972, and 3136 — are four of the five universal outlier channels from the activation heatmap. The fifth (639) drops below the 80% threshold here because it only becomes an outlier at thresholds below 5.0 in middle layers.

Understand why 6.0 is the right threshold

The threshold 6.0 in Listing 7.4 is not arbitrary. Dettmers et al. found it empirically, and we can see exactly why by sweeping thresholds from 0 to 20 on three layers - early (0), middle (16), and late (31).

Figure 7.6 Threshold sweep for OPT-6.7B. Left: fraction of dimensions routed to FP16 as a function of threshold. Right: error reduction ratio (log scale). The shaded band marks the 5.0-7.0 sweet spot. Each layer reveals a different personality: layer 0 (circles, solid) is threshold-insensitive, layer 16 (squares, dashed) shows the clearest knee at 6.0, and layer 31 (triangles, dash-dot) has broadly elevated activations requiring more outlier routing at any threshold.



The sweep reveals three distinct layer personalities. Layer 0 (early) is threshold-insensitive: from 4.0 to 10.0, it flags the same 7 dimensions with the same 4,525× error reduction. The outliers are so extreme (magnitudes above 60, per Figure 7.2) that any reasonable threshold catches them.

Layer 16 (middle) is where the 6.0 default earns its keep. At threshold 2.0, 65% of all 4,096 dimensions are flagged - wasteful and defeating the purpose of INT8 compression. At 6.0, just 0.2% are flagged (8 dimensions) with 172× error reduction - the knee of the curve.

Layer 31 (late) has broadly elevated activations: at threshold 2.0, *all 4,096 dimensions* are outliers. At 6.0, 1% of dimensions are flagged with a modest 14× ratio - diminishing returns but still worthwhile.

The practical insight is that the threshold primarily matters for the middle layers. Early layers are so extreme that any threshold works; late layers have diffuse activations where no single threshold dramatically changes the picture. The middle layers are where threshold tuning determines whether you route 65% of dimensions through FP16 (defeating INT8's purpose) or 0.2% (efficient decomposition). The `llm_int8_threshold` parameter in

bitsandbytes exposes this knob, but 6.0 is well-calibrated for the layers where it matters most.

Deploy with bitsandbytes

Everything in Listings 7.3–7.4 — the outlier detection, column splitting, dual-precision matmul, and summation — is exactly what the bitsandbytes library does in production, wrapped behind two flags in the Hugging Face loader. Listing 7.5 shows the equivalent three-line invocation.

Listing 7.5 LLM.int8() deployment with bitsandbytes

```
from transformers import AutoModelForCausalLM, BitsAndBytesConfig

quantization_config = BitsAndBytesConfig(
    load_in_8bit=True,
    llm_int8_threshold=6.0,
)

model = AutoModelForCausalLM.from_pretrained(
    "facebook/opt-6.7b",
    quantization_config=quantization_config,
    device_map="auto",
)
```

Loading OPT-6.7B with `load_in_8bit=True` replaces all 192 `nn.Linear` layers with `Linear8bitLt` modules that implement the decomposition in CUDA. The model loads in 6.4 GB - a 48% reduction from the 12.4 GB FP16 baseline - and is immediately ready for inference with no calibration step, no retraining, and no model-specific configuration.

Validate with perplexity: the three-way comparison

The reconstruction MSE analysis (Table 7.2) shows that the decomposition reduces per-layer matmul error by orders of magnitude. But does that matmul-level improvement survive stacking through 32 layers and translate into end-to-end language modeling quality? We compare three configurations on WikiText-2 test with 64 sequences of length 512.

Table 7.3 Perplexity comparison on WikiText-2 test (OPT-6.7B)

Method	Perplexity	Δ vs FP16	GPU Memory
FP16 baseline	16.31	—	12.4 GB
Naive INT8 (W+A, per-tensor)	48.35	+32.04	12.4 GB*
LLM.int8()	16.31	+0.00	6.4 GB

NOTE

Why does the naive INT8 row show 12.4 GB? The naive baseline is a quality simulation, not a deployment path. We quantize weights per-channel and activations per-tensor (using forward hooks that quantize-dequantize at every Linear layer), but PyTorch still stores the model weights in FP16 because standard nn.Linear cannot perform real INT8 matmuls. The memory column reflects this simulation overhead. A real INT8 deployment - like the Linear8bitLt modules in bitsandbytes - stores weights as actual int8 and uses ~6.4 GB. The point of this row is perplexity, not memory: naive per-tensor activation quantization triples the perplexity regardless of how the weights are stored.

The naive INT8 row is scale-poisoning end-to-end. Per-tensor activation quantization forces a single scale across all 4,096 hidden dimensions. When a magnitude-60 channel in dimension 1778 sets that scale, the step size becomes 60/127 ≈ 0.47, leaving the 99% of channels with magnitudes below 3 only ~6 usable INT8 levels. The information is not clipped - it is crushed to zero by insufficient resolution. Stacked across 32 layers, this turns a perplexity of 16.31 into 48.35: a 196% increase.

LLM.int8() eliminates this entirely. By routing the 7-41 outlier dimensions per layer through a full-precision path, the INT8 scale for the remaining 99.8% of dimensions is set by well-behaved magnitudes below 3, giving the full 127 usable levels. The result is zero perplexity degradation - 16.31 matching FP16 exactly - at half the memory footprint.

The combination tells the deployment story in one sentence: LLM.int8() gives you the quality of FP16 at the memory cost of INT8, with no

calibration data and no retraining, by spending ~0.1% of the matmul compute on full-precision arithmetic for the dimensions that matter most.

Reproducing these results: The companion script ch7/ch7_llm_int8_decomposition.py generates all experiments and figures from this section.

```
# Full pipeline on Colab T4
python ch7_llm_int8_decomposition.py --mode all --save-plots

# CPU-only (runs opt-125m, illustrative only)
python ch7_llm_int8_decomposition.py --mode all --save-plots \
    --model opt-125m --device cpu
```

7.3 Apply Hessian-aware groupwise methods (GPTQ)

The question GPTQ answers: can we be smarter about *which direction* we round each weight, so errors cancel rather than accumulate?

The key turns out to be the **Hessian** — the matrix of second derivatives of the layer's reconstruction error with respect to its weights.

Informally, the Hessian is a sensitivity map: it tells you, for each input dimension, how much the layer's output will swing if you nudge the weights feeding into it. GPTQ uses that map to decide how to round, spending precision on the weights the model actually relies on. Before we implement it, the next subsection builds the intuition from first principles.

The second-order insight: not all weights matter equally

Consider a weight matrix W in a single linear layer, with input activations X from calibration data. The layer's output is Y = WX. When we quantize W to Q, the reconstruction error is:

Equation 7.1

$$\iota = ||WX - QX||_2^2$$

This is the squared difference in the *layer's output*, not in the weights themselves. Minimizing weight-space error (||W − Q||²) treats every weight equally. Minimizing output-space error (||WX − QX||²) prioritizes weights that actually affect what the layer computes—and that depends on the input data.

Think of it like compressing a photograph. Naively reducing every pixel to 16 colors treats the sky and the subject's face identically. A smarter compressor allocates more precision to the face—where human perception is most sensitive—and lets the sky degrade. The "sensitivity map" that tells you which pixels matter most is analogous to what the Hessian provides for weights.

The Hessian matrix H captures exactly this sensitivity. For the layer reconstruction objective, H is the second derivative of the loss with respect to the weights, and it has a clean closed form:

Equation 7.2

$$H = XX^T$$

where X is the matrix of calibration activations flowing through the layer. The diagonal entry H_j tells you how sensitive the output is to perturbations in the j-th input dimension: a large H_j means that rounding the weights connected to dimension j will cause a large output error. A small H_j means the output barely notices.

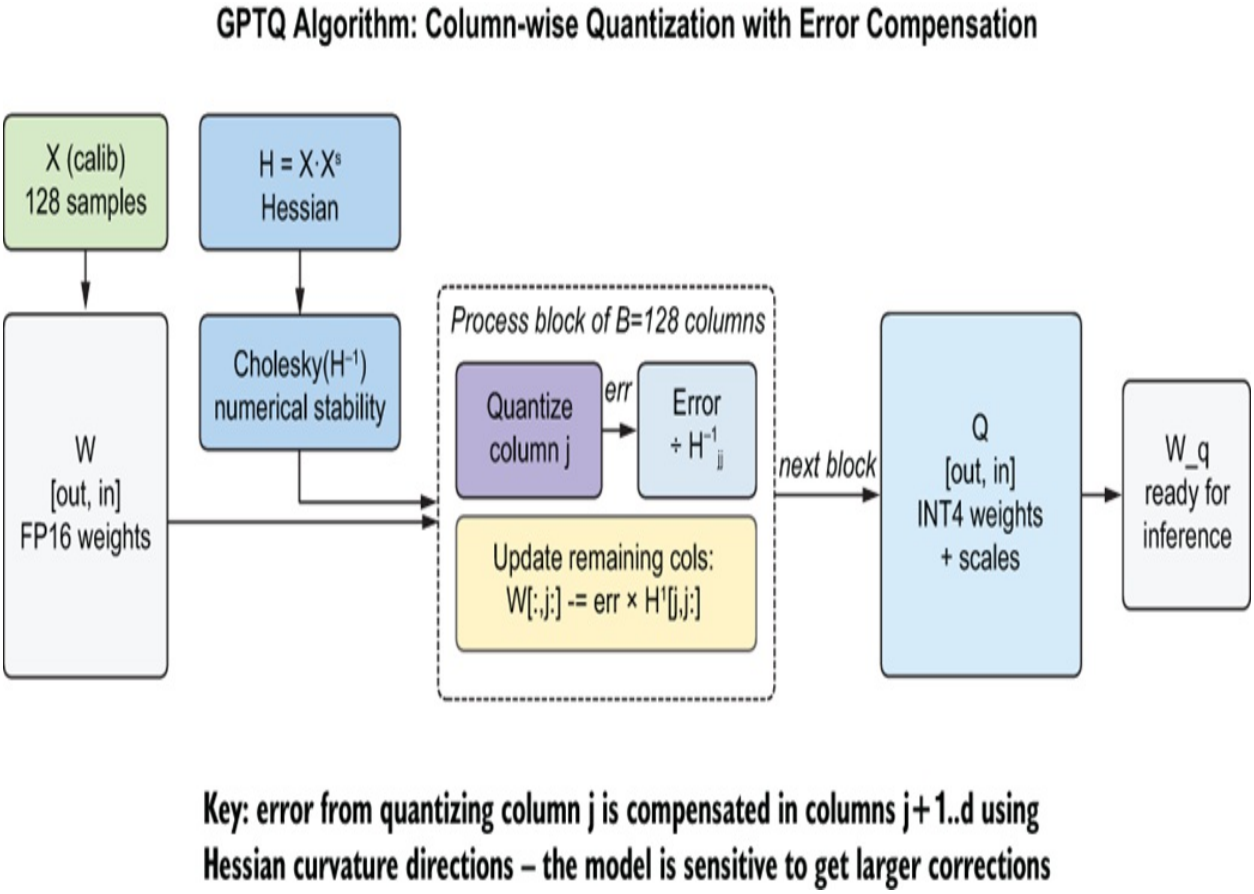
This is not an exotic quantity. If you have run calibration data through a model to collect activations—which every PTQ method in Chapter 4 already does—then $H = XX^T$ is a single matrix multiply away. The entire GPTQ algorithm is built on this one extra computation.

The algorithm: column-wise quantization with error compensation

GPTQ (Frantar et al., ICLR 2023) quantizes a weight matrix one column at a

time, left to right. After quantizing column j, the rounding error is not discarded—it is *distributed* across the remaining columns j+1 through d, weighted by how much the model cares about each direction (the Hessian). Figure 7.7 illustrates the full GPTQ algorithm end-to-end.

Figure 7.7 GPTQ algorithm flow. Calibration activations X produce the Hessian $H = XX^T$, which is Cholesky-factored for numerical stability. The weight matrix W is processed in blocks of B=128 columns. Within each block, each column is quantized to INT4, and the rounding error is distributed across remaining columns using the inverse Hessian. The output is the quantized weight matrix Q plus per-group scales.



Suppose you need to round every department's budget to the nearest thousand dollars. If you round the marketing budget down by $400, you don't just accept the shortfall—you add $400 to the next department's budget *before* rounding it, weighted by how much that department's spending correlates with marketing's impact. Departments that serve the same customers (high Hessian cross-terms) get larger corrections; independent departments get smaller ones. By the time you've rounded every line item, the total budget

deviation is far smaller than if you had rounded each independently.

Three engineering innovations make this practical for billion-parameter models:

- **Fixed column order** instead of greedy selection. The original Optimal Brain Quantization (OBQ) method picked the "easiest" weight to quantize at each step. GPTQ processes columns in sequential order, which means the inverse Hessian can be shared across all rows of the weight matrix—reducing the per-layer cost from $O(d^3 d_{out})$ to $O(d^3)$.
- **Lazy batch updates** of B=128 columns. Rather than propagating error corrections to all remaining columns after every single column quantization, errors are accumulated within a block and applied in one batch matrix multiply at the block boundary. This converts d sequential GPU operations into d/B batched ones.
- **Cholesky decomposition** of H^{-1}. Instead of directly inverting the Hessian (numerically unstable for near-singular matrices), GPTQ precomputes the Cholesky factor of H^{-1} and reads off diagonal entries as needed. A small dampening term $\lambda = 1\%$ of the mean diagonal stabilizes the factorization—a numerical-stability default applied uniformly across architectures rather than a per-model knob.

Implement GPTQ from scratch

Listing 7.6 implements the core algorithm on a single linear layer's weight matrix. The inputs are the weight matrix W [out_features, in_features] and the Hessian H [in_features, in_features] computed from calibration activations.

Listing 7.6 GPTQ quantization (Algorithm 1, Frantar et al. 2023)

```
def gptq_quantize(W, H, bits=4, blocksize=128, groupsize=-1,
                  damp_percent=0.01):
    W = W.float().clone()
    rows, cols = W.shape
    maxq = 2 ** bits - 1

    # Step 0: Cholesky of inverse Hessian
    H = H.float().clone()
```

```
    dead = torch.diag(H) == 0                                   #A
    H[dead, dead] = 1.0

    damp = damp_percent * torch.diag(H).mean()                  #B
    H.diagonal().add_(damp)

    L = torch.linalg.cholesky(H)
    Hinv = torch.cholesky_inverse(L)
    Hinv = torch.linalg.cholesky(Hinv, upper=True)              #C

    Q = torch.zeros_like(W)

    for block_start in range(0, cols, blocksize):
        block_end = min(block_start + blocksize, cols)
        Err = torch.zeros(rows, block_end - block_start,
                          device=W.device, dtype=W.dtype)

        for j in range(block_start, block_end):
            w_col = W[:, j]
            d = Hinv[j, j]                                      #D

            q_col = quantize_column(w_col, scale, zero, maxq)
            Q[:, j] = q_col

            err = (w_col - q_col) / d                           #E
            Err[:, j - block_start] = err

            W[:, j:block_end] -= (
                err.unsqueeze(1) * Hinv[j, j:block_end]         #F
            )

        W[:, block_end:] -= Err @ Hinv[block_start:block_end,
                                       block_end:]              #G

    return Q
```

The critical line is #F: after quantizing column j, the error is not just recorded—it is pushed into columns j+1 through block_end, scaled by the off-diagonal entries of H^{-1}. These entries encode the correlations between input dimensions. If dimensions j and k are strongly correlated in the calibration data (large H^{-1}_{jk}), rounding error in j gets a large correction in k. If they are independent (small H^{-1}_{jk}), the correction is negligible. The Hessian is the reason GPTQ can push to 4 bits without the quality loss that RTN suffers.

Measure sensitivity with the Hessian diagonal

Before comparing full quantization results, the Hessian diagonal itself reveals the sensitivity landscape. Each entry $H_j = \Sigma_n x^2_{nj}$ is the sum of squared activations in dimension j across all calibration samples—a direct measure of how much the model's output depends on weights connected to that input feature.

Listing 7.7 computes the Hessian from captured layer activations.

Listing 7.7 Computing the Hessian from calibration activations

```
captured_inputs = []

def capture_hook(module, inp, out):
    x = inp[0].detach()
    captured_inputs.append(
        x.reshape(-1, x.shape[-1]).cpu()                    #A
    )

hook = target_module.register_forward_hook(capture_hook)

with torch.no_grad():
    for seq in calib_seqs[:8]:                              #B
        model(seq.to(device))
hook.remove()

X = torch.cat(captured_inputs, dim=0).T.float()            #C
H = (X @ X.T) / X.shape[1]                                 #D
```

Figure 7.8 (left panel) plots the Hessian diagonal for layer 0's fc1 (4,096 input features). Most dimensions cluster around a baseline value of 0.03–0.05, but a handful spike to 100–1,000×. The spikes occur at dimensions 639, 1778, 2394, 2972, and 3136 — the same channels that dominated the activation heatmap, now viewed through a different lens. The equivalence is mathematical: $H_j = \Sigma\, x_j^2$, so a large activation magnitude on dimension j produces a large Hessian diagonal at j.

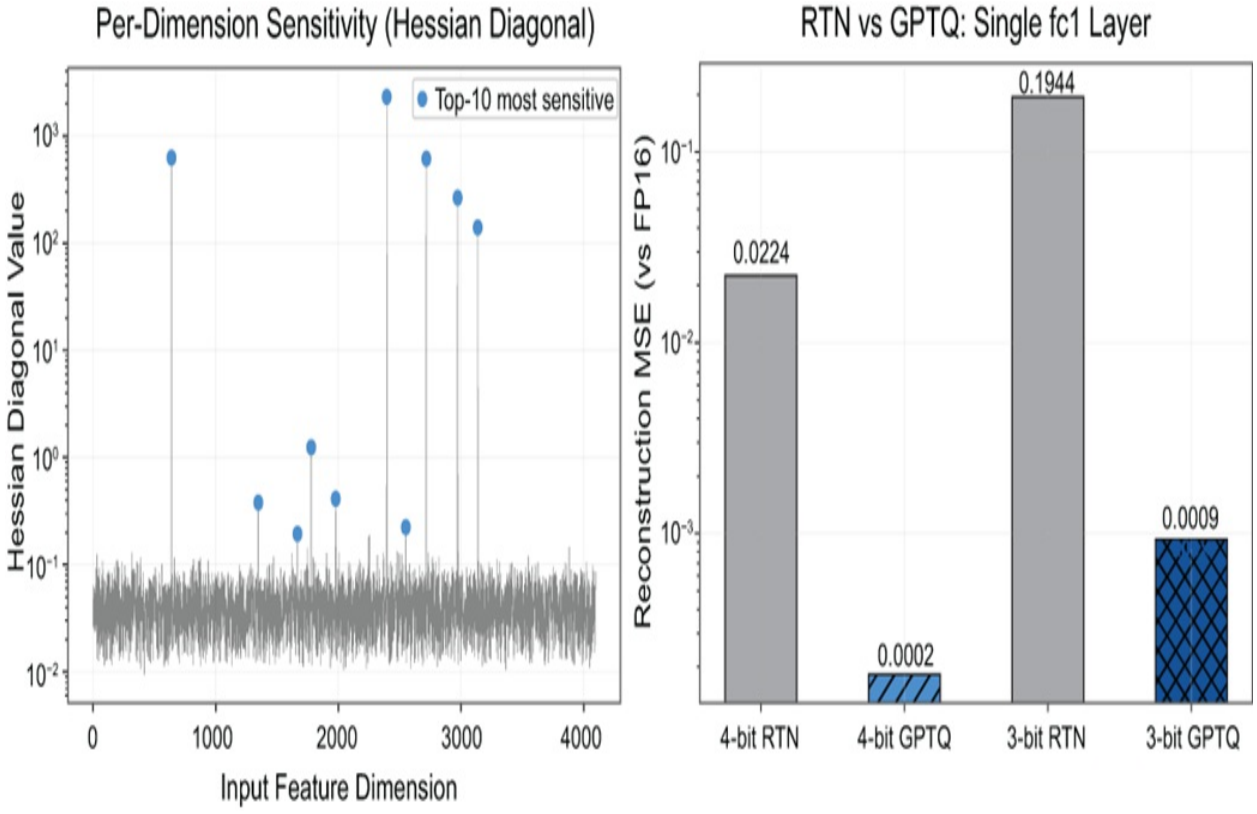
The practical consequence: RTN treats all 4,096 dimensions identically when rounding. GPTQ gives the high-H_j dimensions larger error corrections,

effectively saying "be more careful rounding the weights that connect to dimensions the model relies on most."

Confirm the advantage on a single layer

The right panel of Figure 7.8 shows what this buys us. We quantize layer 0's fc1 weight matrix (shape [16384, 4096]) to both 4-bit and 3-bit using RTN and GPTQ, then measure reconstruction MSE—the squared error between the FP16 output $Y_{ref} = WX$ and the quantized output $Y_q = QX$, averaged over calibration tokens.

Figure 7.8 Left: Hessian diagonal for OPT-6.7B layer 0 fc1. Most of the 4,096 input dimensions have low sensitivity (baseline ~0.04), but the top-10 dimensions spike to 100–1,000×, matching the outlier channels from Section 7.1. Right: reconstruction MSE for RTN vs GPTQ on this single layer. At 4-bit, GPTQ reduces error by 112× (0.0224 → 0.0002). At 3-bit, the advantage grows to 216× (0.1944 → 0.0009)—Hessian-aware error compensation becomes more valuable as the grid gets coarser.



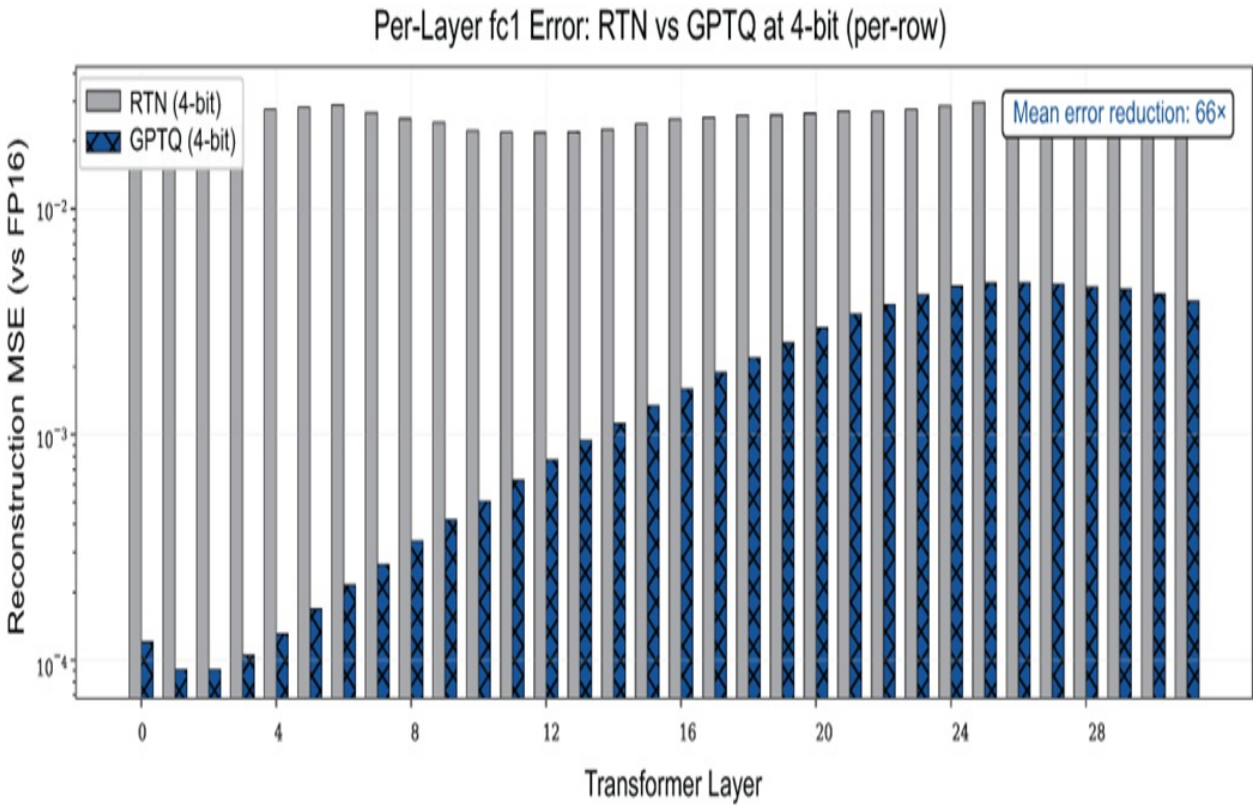
At 4-bit, GPTQ reduces reconstruction MSE from 0.0224 (RTN) to 0.0002—

a 112× improvement on a single layer. At 3-bit, the gap widens to 216×: RTN's MSE jumps to 0.1944 while GPTQ holds at 0.0009. This pattern is important: the coarser the quantization grid, the more rounding error there is to compensate, and the more GPTQ's error distribution mechanism matters. At INT8 (256 levels), RTN is already good enough. At INT4 (16 levels), it is not. At INT3 (8 levels), it is dramatically not.

Scale to every layer in the model

A single-layer result could be an anomaly—perhaps layer 0 is unusually friendly to Hessian-aware quantization. Figure 7.9 extends the comparison to all 32 fc1 layers in OPT-6.7B, quantizing each to 4-bit per-row.

Figure 7.9 Per-layer fc1 reconstruction MSE for RTN vs GPTQ (cross-hatched) at 4-bit per-row quantization across all 32 transformer layers in OPT-6.7B. GPTQ reduces error in every layer, with a mean improvement of 66×. Both methods show increasing MSE in later layers, but GPTQ's advantage holds throughout.



GPTQ wins in every single layer, with a mean error reduction of 66×. Two

patterns stand out. First, both RTN and GPTQ errors increase from early to late layers, mirroring a now-familiar shape: later layers have more diffuse activation distributions, making quantization harder for any method.

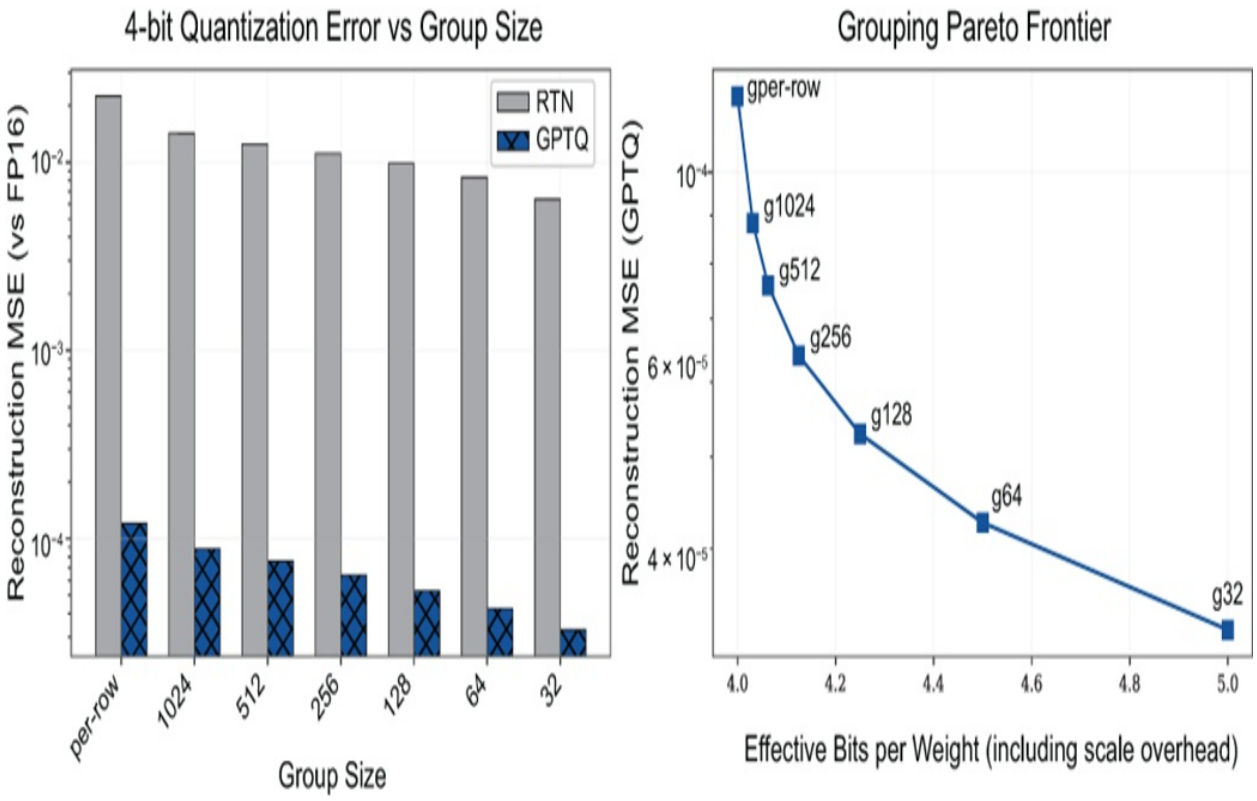
Understand why group size 128 is the default

The per-row results above use a single scale and zero-point per output channel — each scale covers all 4,096 input features. Group quantization tightens this: partition the input dimension into groups of G elements and give each group its own scale. Smaller groups buy finer-grained quantization at the cost of more metadata overhead.

For GPTQ, grouping interacts especially well with the algorithm. At each group boundary, GPTQ recomputes the scale and zero-point using the *current* (error-compensated) weights, not the original weights. This means the quantization grid adapts to the corrections that earlier columns have already applied—a compounding benefit that RTN cannot exploit because it has no error compensation.

Figure 7.10 sweeps group sizes from per-row through 32 on layer 0's fc1.

Figure 7.10 Left: reconstruction MSE at different group sizes for RTN and GPTQ (cross-hatched). Both methods improve with smaller groups, but GPTQ improves faster—its advantage over RTN grows as groups get finer. Right: GPTQ's Pareto frontier of reconstruction MSE vs effective bits per weight (including the FP16 scale and zero-point overhead per group). Group size 128 sits at the knee—further shrinking to 64 or 32 yields diminishing MSE returns at increasing storage cost.



The left panel shows both methods improving with smaller groups, but GPTQ's advantage widens: at per-row, the ratio is ~36×; at group-32, it exceeds 200×. The right panel reveals why 128 is the community default. Each group stores one FP16 scale + one FP16 zero-point = 32 bits of overhead, amortized over G × 4 bits of weight data. At g128, this adds 0.25 bits per weight (4.25 effective), and MSE drops substantially from per-row. At g64, the overhead doubles to 0.5 bits (4.5 effective) for a modest further MSE reduction. At g32, you're spending nearly 5 effective bits per weight—approaching INT5 territory—for diminishing returns.

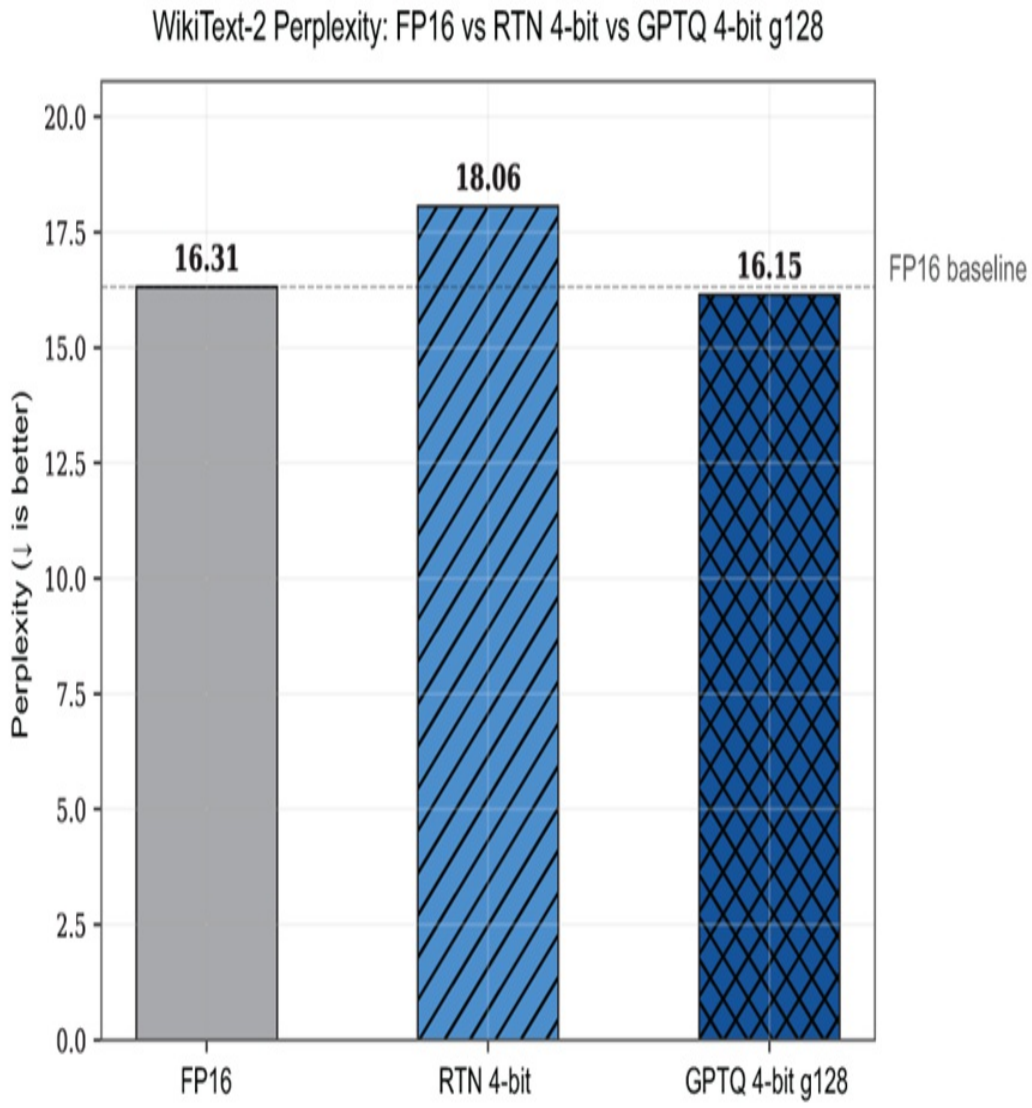
The trade-off is concrete: g128 gives you most of the grouping benefit at ~6% storage overhead. Production tools like gptqmodel, llama.cpp, and ExLlamaV2 all default to g128 for this reason.

The perplexity showdown: does layer-level accuracy survive stacking?

Reconstruction MSE on individual layers is necessary but not sufficient. Errors compound through 32 layers of sequential computation—a 66× per-layer advantage could shrink, hold, or even amplify at the model level. The

definitive test is end-to-end perplexity on held-out text. We evaluate three configurations on WikiText-2 test (64 sequences of length 512, OPT-6.7B on a T4 GPU): FP16 baseline, RTN 4-bit per-row (simulated by quantize-dequantize on all 193 Linear layers), and GPTQ 4-bit with group size 128 via gptqmodel. Figure 7.11 shows the result.

Figure 7.11 WikiText-2 perplexity for OPT-6.7B. FP16 baseline (16.31) compared to RTN 4-bit per-row (hatched, 18.06) and GPTQ 4-bit g128 (cross-hatched, 16.15). The dashed line marks the FP16 baseline. RTN degrades perplexity by 10.7%. GPTQ avoids degradation entirely—scoring 0.16 points below FP16 (within measurement noise at this evaluation length) while loading in 3,626 MB vs FP16's 12,700 MB.



NOTE

The RTN result is a quality simulation, not a deployment path. We quantize and dequantize all Linear layer weights, but PyTorch still stores them as FP16—hence both FP16 and RTN show ~12,700 MB. A real 4-bit deployment stores packed INT4 weights and uses ~3.6 GB, as the GPTQ row demonstrates.

RTN 4-bit adds 1.75 perplexity points—a 10.7% increase that is noticeable in generation quality. When a language model's perplexity rises by this much, outputs become measurably less fluent: sentences that the FP16 model would have ranked highly get suppressed, and lower-quality alternatives rise in their place.

GPTQ 4-bit g128 achieves 16.15—0.16 points *below* the FP16 baseline. This is within evaluation variance (our 64 × 512-token evaluation window introduces sampling noise on the order of ±0.2 points), so the result should be read as "no measurable degradation" rather than "GPTQ improves quality." Hessian-aware error compensation eliminates the entire 1.75-point RTN penalty while compressing the model from 12,700 MB to 3,626 MB—a 3.5× reduction that puts OPT-6.7B comfortably on a single 8 GB consumer GPU.

Deploy with gptqmodel

The from-scratch implementation in Listing 7.6 teaches the algorithm, but production quantization uses optimized libraries. gptqmodel (the actively maintained successor to the archived auto-gptq) handles Hessian computation, layer-sequential processing, INT4 weight packing, and serialization in a single API.

Listing 7.8 GPTQ production deployment with gptqmodel

```
from gptqmodel import GPTQModel, QuantizeConfig

quantize_config = QuantizeConfig(
    bits=4,                                                    #A
    group_size=128,                                            #B
    damp_percent=0.01,                                         #C
    desc_act=False,                                            #D
)
```

```
model = GPTQModel.load("facebook/opt-6.7b", quantize_config)
model.quantize(calib_data)                                    #E
model.save_quantized("./opt-6.7b-gptq-4bit")                  #F
```

Quantizing OPT-6.7B on a T4 GPU takes approximately 28 minutes. The saved model is 3.6 GB—a 71.6% reduction from the 12.7 GB FP16 checkpoint. Loading the quantized model for inference requires two lines:

```
model = GPTQModel.from_quantized(
    "./opt-6.7b-gptq-4bit",
    device="cuda:0",
)
```

The quantized model is immediately usable for generation with the standard transformers API. No additional calibration, no runtime overhead beyond the INT4 → FP16 dequantization that happens inside each linear layer, and no changes to the inference pipeline.

NOTE

desc_act=False vs desc_act=True. The desc_act parameter controls whether columns are processed in descending order of Hessian diagonal (activation-ordered) rather than sequentially. Activation ordering can slightly reduce perplexity on some models because it quantizes the most sensitive columns first, when there are the most remaining columns to absorb error. The cost is that the weight matrix must be permuted at inference time, preventing some kernel optimizations. Most practitioners leave it off—the sequential order in our implementation and in the default config gives near-identical quality with better inference compatibility.

What GPTQ does and does not solve

GPTQ is a weight-only method. It quantizes the static parameters of the model — not activations, not the KV cache. During inference, activations still flow through in FP16 (or BF16), and the KV cache consumes memory according to the serving-capacity numbers in Table 7.1.

GPTQ also requires calibration data; 128 sequences is the standard default.

The data should be representative of the deployment domain, but the method is relatively robust to the specific choice: the Hessian captures second-order statistics (correlations between input features), which are more stable across text distributions than first-order statistics like means or maxima. A Wikipedia sample works for most general-purpose LLM quantization.

The main limitation is quantization time. The 28-minute cost for OPT-6.7B scales roughly linearly with parameter count — a 70B model takes several hours. This is a one-time offline cost (quantize once, serve forever), but it is also where LLM.int8() wins on time-to-deploy: that method takes zero calibration and runs at load time.

Reproducing these results: The companion scriptch7/ch7_gptq_quantization.py generates all experiments and figures from this section.

```
# Full pipeline on Colab T4
python ch7_gptq_quantization.py --mode all --save-plots

# CPU-only (uses OPT-125M, illustrative only)
python ch7_gptq_quantization.py --mode all --save-plots \
    --model opt-125m --device cpu
```

7.4 Apply activation-aware weight protection (AWQ)

GPTQ solves the weight quantization problem by correcting errors after they happen—quantizing a column, measuring the damage and redistributing it across remaining columns using the Hessian. It works beautifully, but it requires computing and factoring a second-order matrix for every layer. AWQ (Lin et al., MLSys 2024) asks a different question: what if we could prevent the worst errors from happening in the first place?

The intuition is medical. GPTQ is surgery—skilled, precise, but invasive. It opens up the weight matrix, cuts (quantizes), and sutures (compensates) column by column. AWQ is vaccination—it strengthens the vulnerable channels before quantization ever touches them, so that standard round-to-nearest produces a good result without any error compensation. No Hessian,

no Cholesky factorization, no sequential column processing. Just a per-channel scaling trick that makes the quantization grid finer where it matters most.

The key insight: not all weight channels are equally important for output quality, and the ones that matter most are identified by the *activations* that multiply them—not by the weights themselves.

Identify which channels matter: activation magnitude, not weight magnitude

Consider a single linear layer computing y = Wx. Each column of W multiplies one element of x. If column j has a small weight (say, 0.001) but the activation x_j is large (say, 1,000), the output contribution $w_j x_j = 1.0$ is significant. Quantizing that small weight carelessly—rounding 0.001 to 0.0—destroys a meaningful signal. Conversely, a large weight multiplied by a near-zero activation contributes nothing; quantizing it poorly is free.

This means the weight magnitude alone is a poor predictor of which channels need protection. The activation magnitude is what matters. To verify this empirically, we quantize each of OPT-6.7B's 4,096 input channels independently in layer 0's fc1, measure the per-channel contribution to output error, and ask: does the activation ranking or the weight ranking better predict which channels cause the most damage?

Listing 7.9 shows how to compute the activation importance signal that AWQ uses.

Listing 7.9 Computing per-channel activation importance

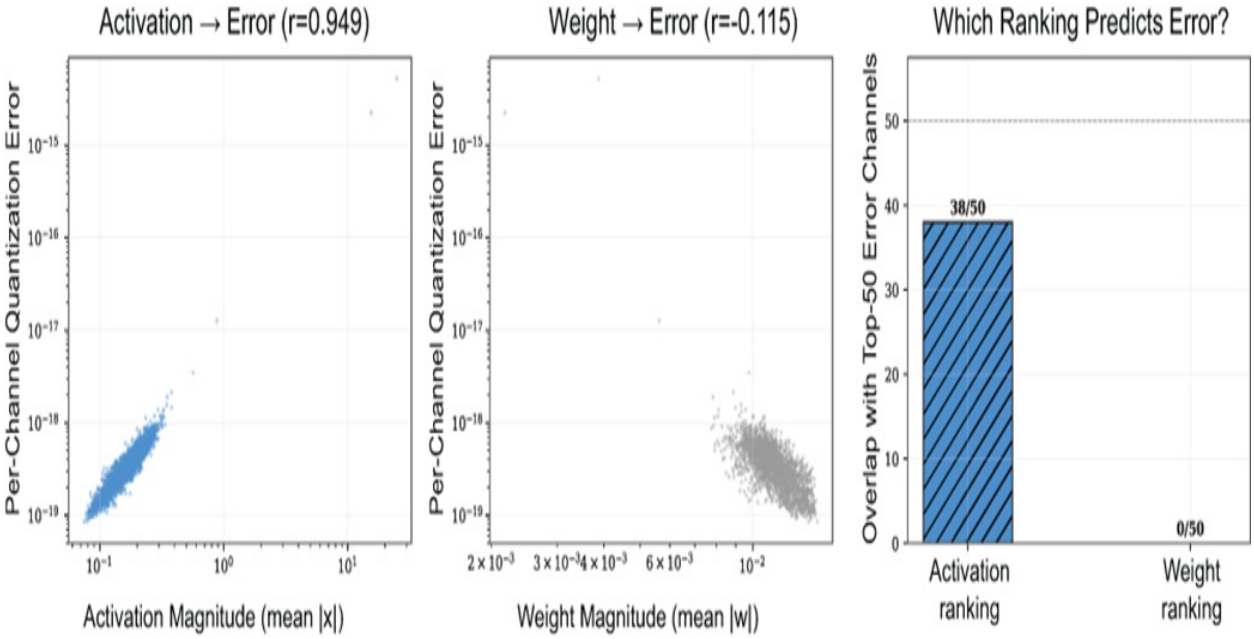
```
def compute_activation_scales(X):
    """Per-channel importance = mean(|x|) across calibration samp
    act_scales = X.abs().mean(dim=0)              #A
    return act_scales                             #B
```

The function is deliberately simple—a single line of computation. AWQ's power comes not from a complex importance metric but from what it *does*

with this signal.

Figure 7.12 shows the result on OPT-6.7B layer 0 fc1, using 128 calibration sequences from WikiText-2.

Figure 7.12 Which ranking predicts quantization error? Left: activation magnitude vs per-channel quantization error (r = 0.949, strong positive correlation). Center: weight magnitude vs the same error (r = −0.115, no useful correlation). Right: of the 50 channels with the highest true quantization error, activation ranking identifies 38 of them; weight ranking identifies zero. Activation magnitude is the correct importance signal for weight quantization.



The correlation is striking: r = 0.949 for activation magnitude versus r = −0.115 for weight magnitude. Of the 50 channels that cause the most output error when quantized, the activation ranking catches 38—the weight ranking catches none. Weight magnitude is not just a weak predictor; it is essentially uncorrelated with quantization damage.

This result is counterintuitive. We are quantizing *weights*, yet the *activations* determine which weight columns matter. The reason is the multiplicative structure of the linear layer: quantization error on weight column j is amplified by activation magnitude x_j before reaching the output. A 1% error on a weight multiplied by 1,000 produces 10× more output damage than a 10% error on a weight multiplied by 0.01.

The equivalent transformation: scale, quantize, absorb

Now that we know which channels matter, the question is what to do about it. AWQ's answer is an equivalent transformation that makes salient weight channels larger before quantization, so they land on finer grid points.

The mathematical core is a per-channel scaling that preserves the layer's output exactly:

Equation 7.3

$$Y = W.X = \left(W.diag\left(s\right)\right).\left(diag(s)^{-1}.X\right) = W^{'}X^{'}$$

where s is a per-channel scale vector. This is an identity—no approximation. The matrix product WX is unchanged because the scaling and its inverse cancel. The trick is that we now quantize W' = W diag(s) instead of W:

Equation 7.4

$$Y \approx Q\left(W.diag\left(s\right)\right).\left(diag(s)^{-1}.X\right)$$

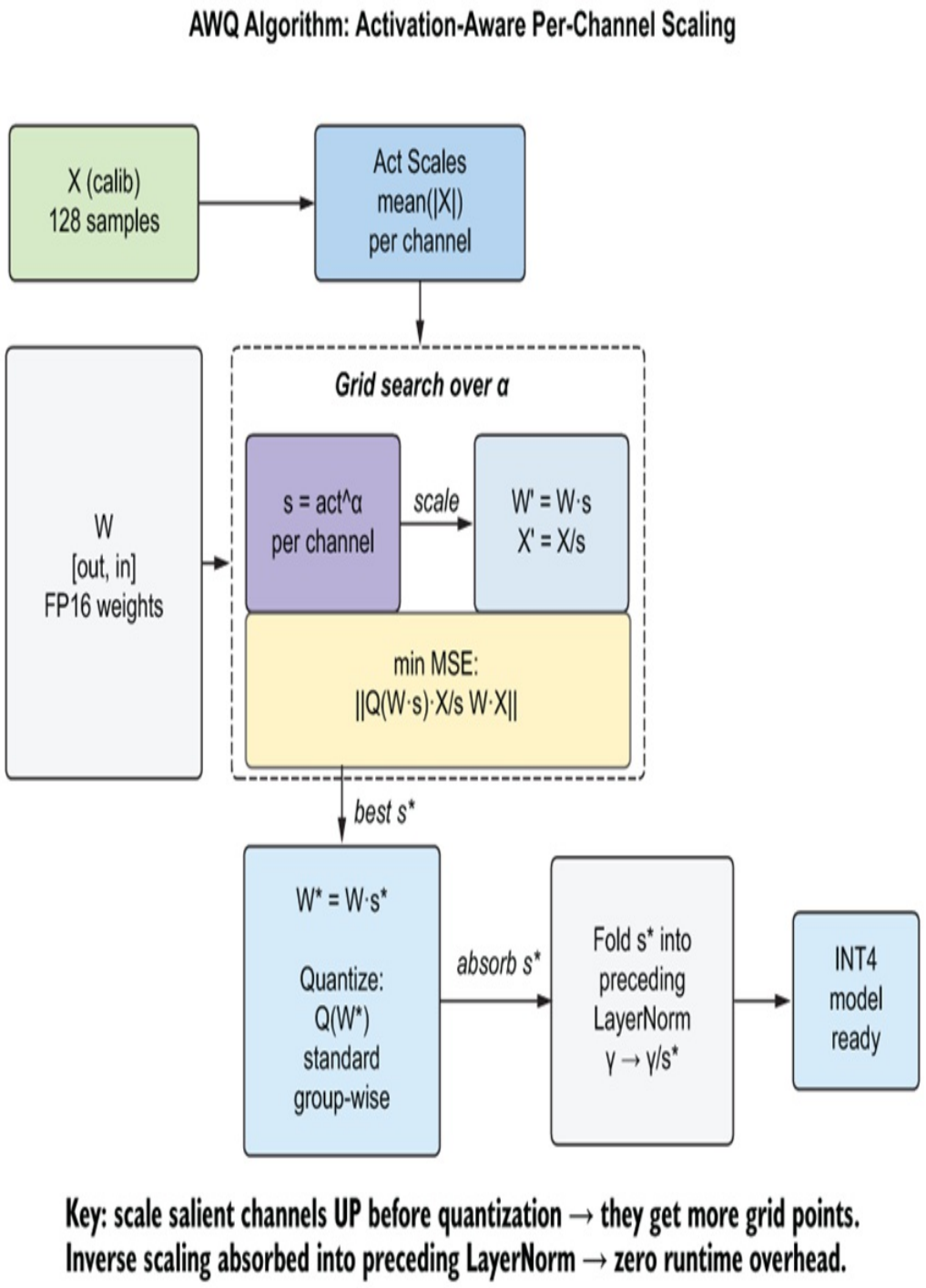
For channels where $s_j > 1$, the weight column is scaled up. This increases the magnitude of the weights in that column, which could in principle inflate the quantization step size Δ = range / $(2^b - 1)$ for the group—but the paper shows this rarely happens. In OPT-6.7B at s = 2, only 8.2% of groups see their step size change at all, and the average increase is just 3.8%. The group maximum is determined by the worst-case element across 128 channels, and one channel's scaling rarely displaces it. Because $\Delta' \approx \Delta$ while the output error is divided by s_j (from the activation-side rescaling), the net effect is approximately a $1/s_j$ reduction in quantization error for the protected channel —clean and predictable.

The key question is where the inverse scaling 1/s goes at inference time. The answer is that it is folded into the preceding operator—in transformers, typically a LayerNorm. The input to each linear layer passes through a LayerNorm with a learned gain parameter γ. Replacing γ with γ/s absorbs the

inverse scaling permanently—the LayerNorm output is pre-divided by s, so the scaled weights W s receive the correctly adjusted input. This folding happens once during quantization; at inference, there is zero computational overhead. The model is simply a standard INT4 model with modified LayerNorm gains. The same principle applies to any preceding linear operator (the AWQ paper cites SmoothQuant's equivalent transformation for the general case).

Figure 7.13 shows the full AWQ pipeline.

Figure 7.13 AWQ algorithm flow. Calibration data produces per-channel activation scales (top). A grid search over the scaling exponent α finds the best per-channel scale vector s* that minimizes reconstruction error after quantization (center). The optimal scales are applied to weights before standard group-wise quantization, then the inverse is folded into the preceding LayerNorm—producing an INT4 model with zero runtime overhead.



The remaining question is how to choose the scale vector *s*. AWQ parameterizes it as:

Equation 7.5

$$s_j = \left(\frac{act_j}{max(act)} \right)^{\alpha}$$

where actj is the per-channel activation magnitude and α ∈ [0, 1] is a scaling exponent. At α = 0, all scales are 1.0 (no scaling—pure RTN). At α = 1, scales are proportional to activation magnitude (maximum protection). The optimal α balances two competing forces: protecting salient channels (higher α) versus inflating the quantization range for non-salient channels in the same group (lower α).

AWQ finds the best α through a simple grid search: try 20 values, keep whichever minimizes reconstruction MSE for each weight group. Listing 7.10 implements this.

Listing 7.10 AWQ per-channel scale search (Equation 5 from Lin et al.)

```
def awq_scale_search(W, X, bits=4, groupsize=128, n_grid=20):
    act_scales = compute_activation_scales(X)              #A
    Y_ref = W @ X.T                                        #B

    best_scales = torch.ones(W.shape[1], device=W.device)

    for g_start in range(0, W.shape[1], groupsize):        #C
        g_end = min(g_start + groupsize, W.shape[1])
        W_group = W[:, g_start:g_end]
        X_group = X[:, g_start:g_end]
        act_group = act_scales[g_start:g_end]

        best_mse = float("inf")
        best_group_scales = torch.ones(g_end - g_start)

        for i in range(n_grid):                            #D
            alpha = 1.0 - i / n_grid
            act_norm = act_group / (act_group.max() + 1e-8)
            scales = act_norm.pow(alpha).clamp(min=1e-4)

            W_scaled = W_group * scales.unsqueeze(0)       #E
            X_descaled = X_group / scales.unsqueeze(0)     #F

            W_q = quantize_rtn(W_scaled, bits=bits)        #G
            Y_q = W_q @ X_descaled.T

            Y_group_ref = W_group @ X_group.T
```

```
            mse = ((Y_group_ref - Y_q) ** 2).mean().item()

            if mse < best_mse:
                best_mse = mse
                best_group_scales = scales.clone()

        best_scales[g_start:g_end] = best_group_scales

    return best_scales
```

The grid search is embarrassingly simple — no gradients, no matrix factorizations, no iterative optimization. For each group of 128 weight columns, try 20 values of α and keep the best one. The entire search for OPT-6.7B's layer 0 fc1 (a [16384, 4096] matrix with 32 groups) completes in about 2 minutes on a T4 GPU.

The scale vector by itself is not yet a quantized model — it still has to be applied to the weights and the weights have to be rounded. Listing 7.11 wraps both steps into a single function: find the scales, apply them, run group-wise RTN on the scaled weights.

Listing 7.11 Full AWQ quantization with scale application

```
def awq_quantize(W, X, bits=4, groupsize=128, n_grid=20):
    scales = awq_scale_search(W, X, bits, groupsize, n_grid)  #A

    W_scaled = W * scales.unsqueeze(0)                         #B
    W_q_scaled = quantize_rtn(W_scaled, bits=bits,
                              groupsize=groupsize)             #C
    W_q = W_q_scaled / scales.unsqueeze(0)                     #D

    return W_q, scales
```

Step D deserves attention. After quantization, we divide the quantized weights by s to undo the scaling. This is a mathematical convenience for our from-scratch implementation—in production, the 1/s is absorbed into the preceding LayerNorm and never explicitly computed at inference time. The result is identical: at inference, the input arrives pre-scaled by 1/s (via the modified LayerNorm), hits the scaled-and-quantized weights Q(Ws), and the *s* factors cancel.

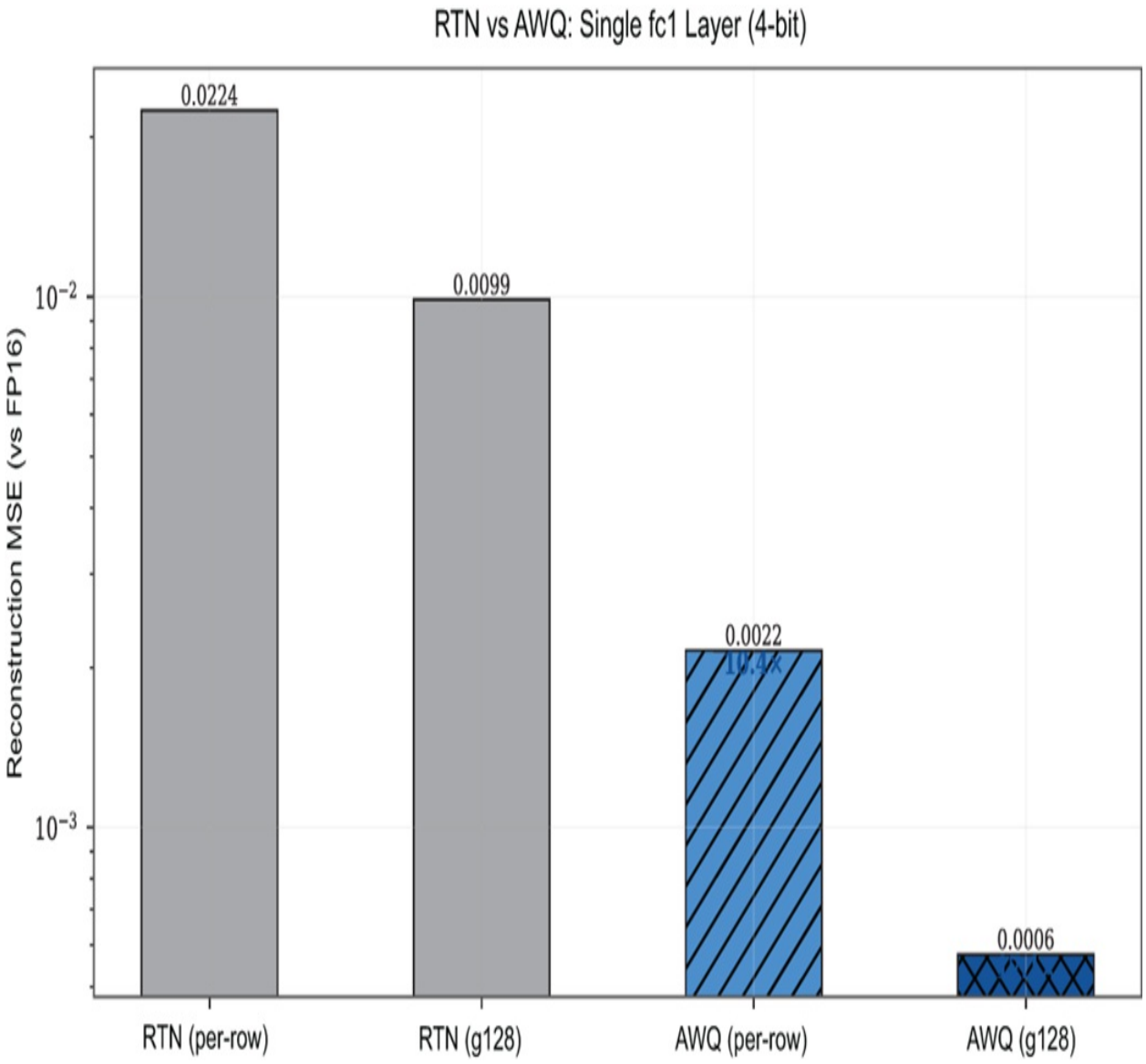
NOTE

The AWQ paper additionally applies per-group weight clipping after scaling —the same MSE-optimal range narrowing from Section 4.3, applied at group granularity. Clipping deliberately sacrifices a few outlier weight values to tighten the quantization range for the majority, reducing granular error at the cost of small clipping error. Our from-scratch implementation omits this step to isolate the effect of activation-aware scaling alone. Production AutoAWQ includes both scaling and clipping; the scaling is the larger contributor to error reduction.

Measure the single-layer improvement

Figure 7.14 compares RTN and AWQ on layer 0's fc1 at 4-bit precision, both per-row and with group size 128.

Figure 7.14 Reconstruction MSE on a single fc1 layer (OPT-6.7B, layer 0, 4-bit). AWQ reduces error by 10.4× with per-row quantization and 17.1× with group size 128. The combination of AWQ scaling and group quantization is multiplicative: group-128 gives RTN a 2.3× boost (0.0224 → 0.0099), but gives AWQ a 3.8× boost (0.0022 → 0.0006) because finer groups interact constructively with per-channel scaling.



The error reductions are large: 10.4× for per-row and 17.1× for group-128. Notice that group quantization and AWQ scaling interact constructively. Grouping alone gives RTN a 2.3× improvement (from 0.0224 to 0.0099). But grouping gives AWQ a 3.8× improvement (from 0.0022 to 0.0006), because smaller groups mean the scale vector s can be optimized at finer granularity —each group of 128 columns gets its own optimal α rather than sharing one α across all 4,096 columns.

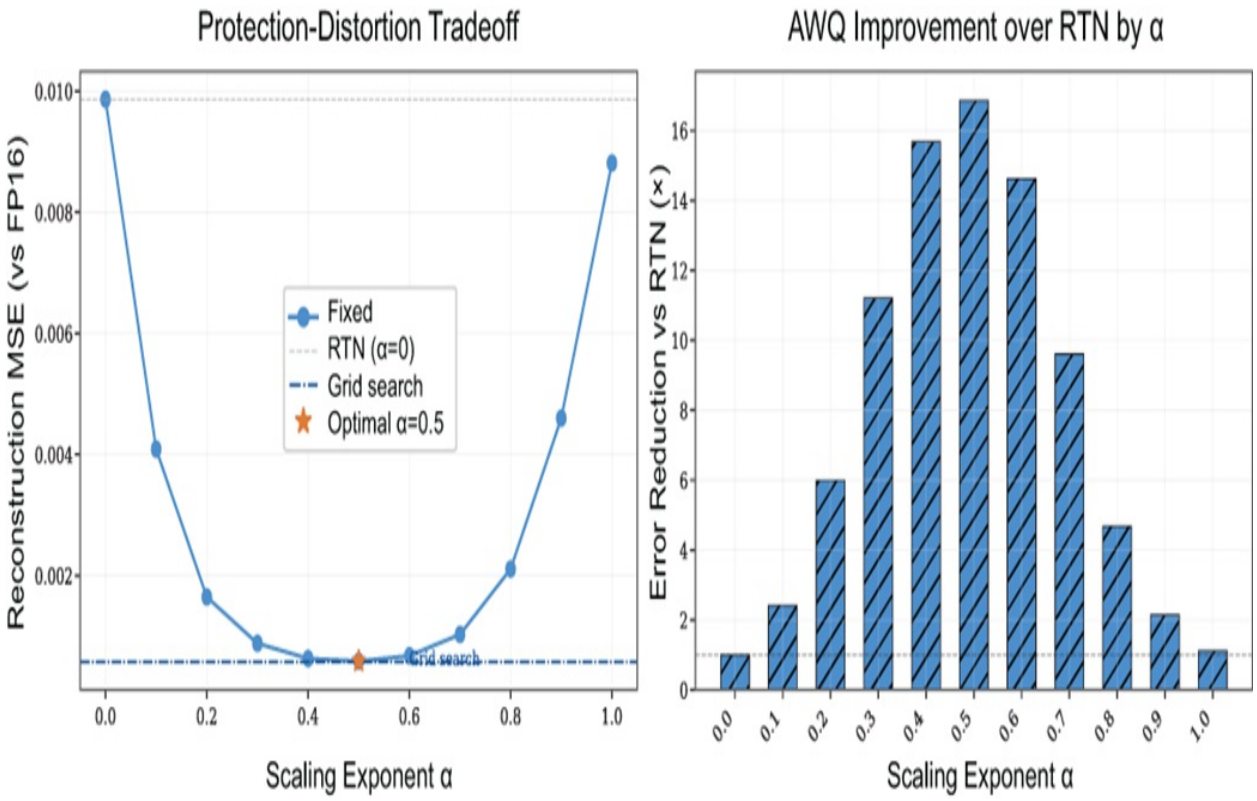
Understand the protection-distortion tradeoff

The scaling exponent α controls a tradeoff: more protection for salient

channels versus more distortion for non-salient channels in the same group. To visualize this, we sweep α from 0.0 (pure RTN) to 1.0 (full activation-proportional scaling) on layer 0's fc1 with group size 128.

Figure 7.15 shows the result.

Figure 7.15 The protection-distortion tradeoff. Left: reconstruction MSE as a function of the scaling exponent α. The curve is U-shaped—too little scaling (α near 0) leaves salient channels unprotected; too much (α near 1) inflates the group scale for non-salient channels. The optimal α = 0.5 achieves 16.9× error reduction versus RTN, nearly matching the full grid search (17.1×, dashed line). Right: error reduction ratio versus RTN at each α. The sweet spot is broad—any α between 0.3 and 0.7 achieves at least 4.7× improvement.



The U-shape reveals the tradeoff clearly. At α = 0, there is no scaling—pure RTN, baseline error. As α increases from 0 to 0.5, salient channels receive progressively more protection, and error drops rapidly: 2.4× at α = 0.1, 11.2× at α = 0.3, 16.9× at α = 0.5. Beyond α = 0.5, the scaling overshoots: the salient channels' magnitudes grow so large that they dominate the group range, inflating the step size Δ for every other channel in the group. By α = 1.0, the damage to non-salient channels almost completely offsets the

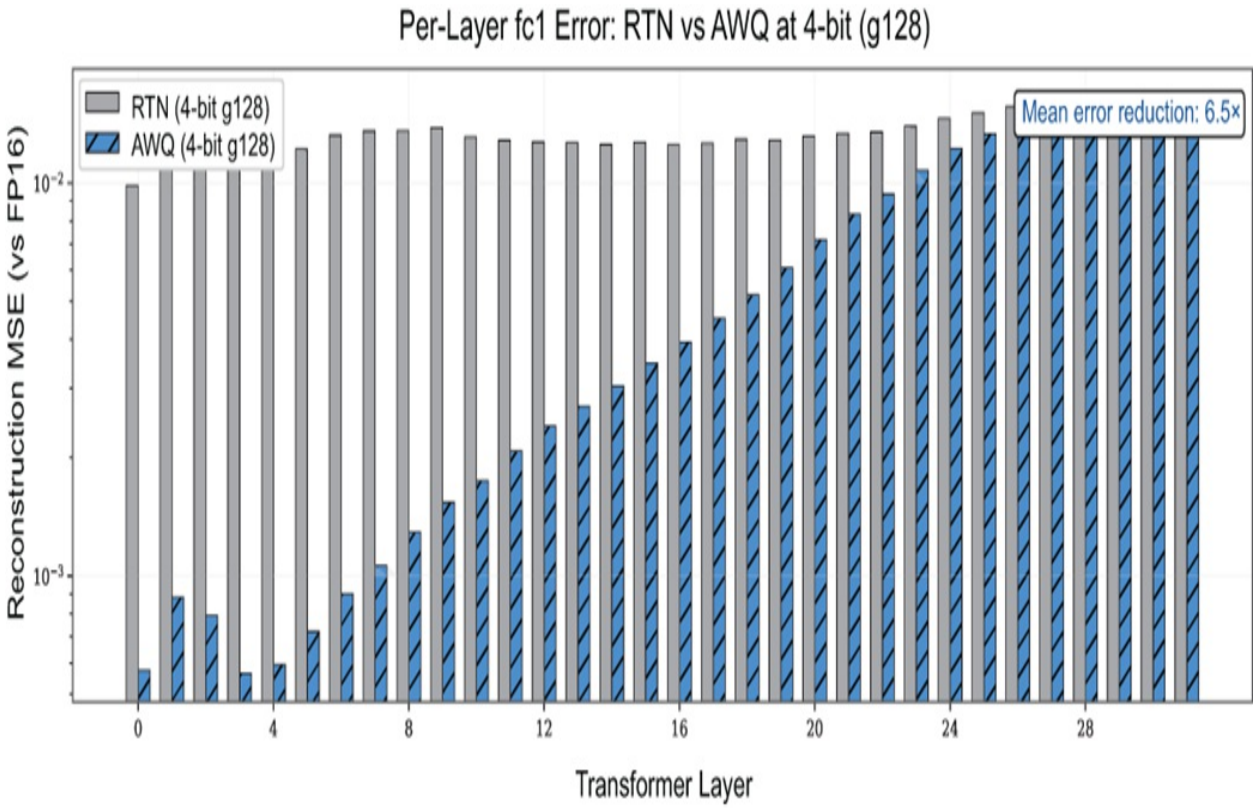
protection of salient ones (only 1.1× improvement).

The optimal α = 0.5 achieves 16.9× error reduction—nearly matching the full per-group grid search (17.1×). This is why AWQ's paper recommends α = 0.5 as the default. The grid search provides a marginal additional benefit by allowing different groups to use different α values, but the fixed default captures almost all of the gain.

Validate across all 32 layers

A single layer is promising but not conclusive. Figure 7.16 extends the comparison to every fc1 layer in OPT-6.7B at 4-bit with group size 128.

Figure 7.16 Per-layer fc1 reconstruction MSE for RTN vs AWQ (hatched) at 4-bit group-128 across all 32 transformer layers in OPT-6.7B. AWQ reduces error in every layer, with a mean improvement of 6.5×. The advantage is largest in early layers (17–21×) and diminishes in later layers (1.1×), mirroring the activation outlier concentration from Section 7.1.



AWQ wins on every single layer, with a mean error reduction of 6.5×. The

pattern is informative: early layers (0–5) show 13–21× improvement, middle layers (10–20) show 2–8×, and late layers (25–31) show only 1.1×. The gradient tracks the same activation-outlier structure we have been measuring all chapter — early layers have the most concentrated, highest-magnitude outlier channels, which gives AWQ's activation-guided scaling the most room to help. In late layers, activation distributions become more diffuse, the distinction between salient and non-salient channels blurs, and AWQ's scaling provides diminishing returns.

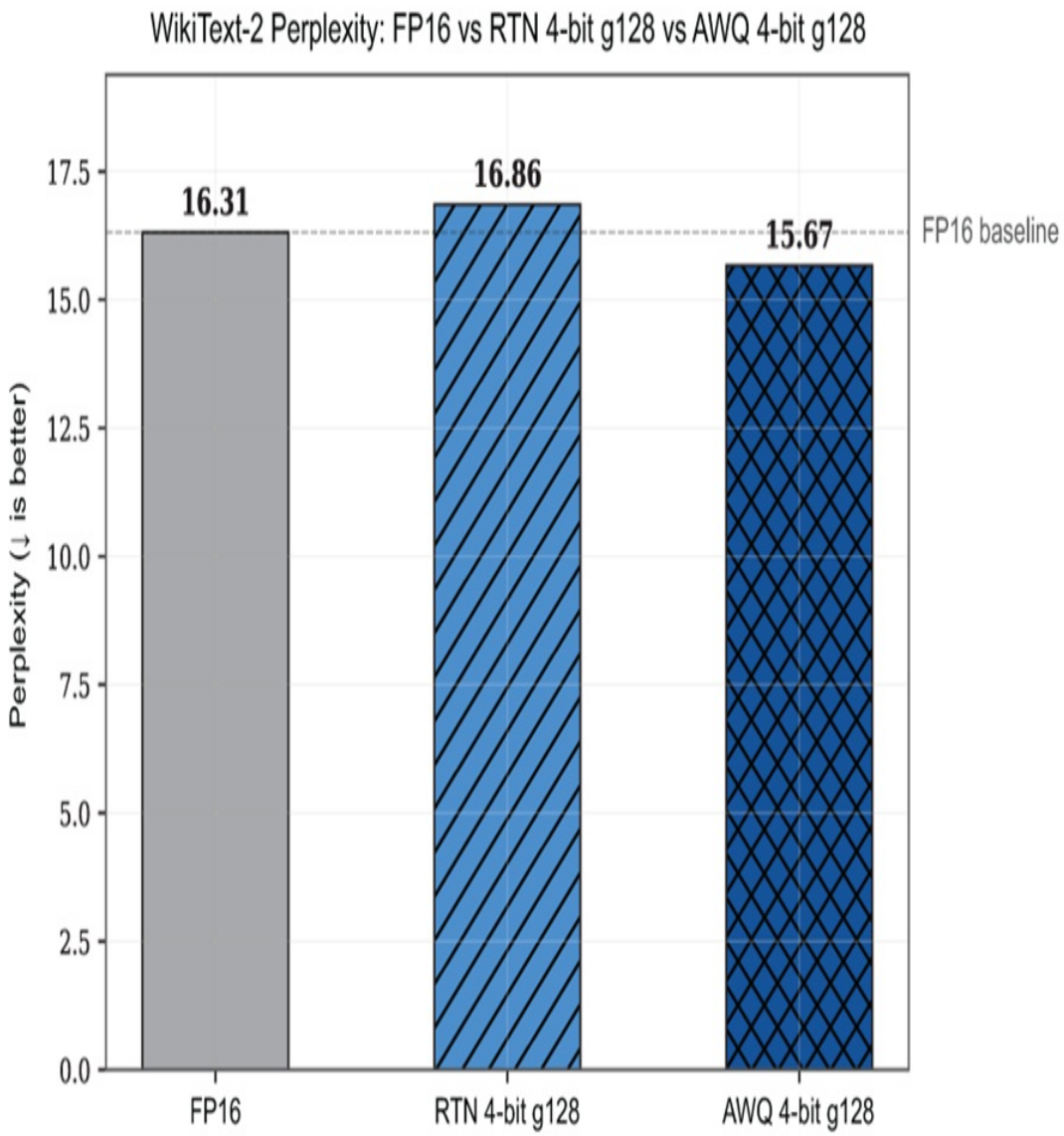
The two methods diverge in the floor of their advantage. GPTQ shows a similar directional pattern, but its error reduction never drops below 10× on any layer. AWQ's advantage drops to 1.1× in the final layers, essentially converging with RTN. The mechanisms explain it: GPTQ's Hessian-guided error compensation adapts to each layer's local error surface regardless of whether outlier structure is present. AWQ's activation-guided scaling depends on a strong contrast between salient and non-salient channels, and that contrast fades as distributions diffuse.

The perplexity showdown: does prevention match correction?

The definitive test is end-to-end perplexity. We compare three configurations on WikiText-2 test (64 sequences of length 512, OPT-6.7B): FP16 baseline, RTN 4-bit group-128 (simulated quantize-dequantize), and AWQ 4-bit group-128 via AutoAWQ—the production library that properly absorbs scales into LayerNorms.

Figure 7.17 shows the result.

Figure 7.17 WikiText-2 perplexity for OPT-6.7B. FP16 baseline (16.31) compared to RTN 4-bit g128 (hatched, 16.86, +0.54) and AWQ 4-bit g128 (cross-hatched, 15.67, −0.64). AWQ not only eliminates the RTN degradation but scores 0.64 points below the FP16 baseline—within evaluation variance but directionally better. The AWQ model loads in 3,613 MB versus FP16's 12,700 MB.



AWQ 4-bit achieves 15.67 perplexity — 0.64 points *below* the FP16 baseline of 16.31. Read this as "no measurable degradation" rather than "AWQ improves quality": the 64 × 512-token evaluation window introduces sampling noise on the order of ±0.5 points. The meaningful comparison is against RTN: AWQ eliminates the 0.54-point RTN penalty entirely, scoring 1.19 points better than RTN 4-bit (15.67 vs 16.86).

The memory savings are real and identical to GPTQ: the AWQ model loads in 3,613 MB versus 12,700 MB for FP16—a 3.5× reduction. Both AWQ and GPTQ produce INT4 models with the same storage format; the difference is how the quantized values are determined, not how they are stored.

NOTE

Comparing AWQ and GPTQ perplexity: AWQ landed at 15.67 on this evaluation; GPTQ at 16.15 on the same setup. The 0.48-point difference favors AWQ but is within evaluation variance — both methods effectively match FP16 quality at 4-bit. The practical difference is in the approach: GPTQ uses second-order optimization (Hessian + Cholesky) and takes 28 minutes for OPT-6.7B; AWQ uses a simple grid search and takes about 32 minutes. Neither requires backpropagation or gradient computation.

Deploy with AutoAWQ

The from-scratch implementation teaches the algorithm, but production quantization uses the AutoAWQ library, which handles the full pipeline: calibration data preparation, per-block scale search, LayerNorm absorption, INT4 weight packing, and serialization.

Listing 7.12 AWQ production deployment with AutoAWQ

```
from awq import AutoAWQForCausalLM
from transformers import AutoTokenizer

model = AutoAWQForCausalLM.from_pretrained(
    "facebook/opt-6.7b", safetensors=False
)
tokenizer = AutoTokenizer.from_pretrained("facebook/opt-6.7b")

quant_config = {
    "zero_point": True,       #A
    "q_group_size": 128,      #B
    "w_bit": 4,               #C
    "version": "GEMM",        #D
}

model.quantize(                #E
    tokenizer,
    quant_config=quant_config,
    calib_data=calib_data,
)

model.save_quantized("./opt-6.7b-awq-4bit")   #F
```

Quantizing OPT-6.7B on an A100 GPU takes approximately 32 minutes. The saved model loads in under 4 GB—compared to 12.7 GB for the FP16 checkpoint. Loading the quantized model for inference takes two lines:

```
model = AutoAWQForCausalLM.from_quantized(
    "./opt-6.7b-awq-4bit",
    fuse_layers=False,
)
```

A quick generation test confirms the model is functional:

```
Prompt: The advantage of AWQ over round-to-nearest quantization i
Output: The advantage of AWQ over round-to-nearest quantization i
        that it is more robust to quantization errors.
```

The quantized model produces coherent, relevant text—and the irony of the model correctly describing its own advantage is a satisfying sanity check.

NOTE

AutoAWQ is deprecated as of early 2025; its maintainer (Casper Hansen) recommends migrating to llm-compressor (maintained by the vLLM project) for new work. The API patterns are similar, and the underlying AWQ algorithm is unchanged. We use AutoAWQ here because it remains the most widely deployed AWQ implementation at the time of writing and the one most readers will encounter in existing codebases.

What AWQ does and does not solve

AWQ, like GPTQ, is a weight-only method. It quantizes static model parameters — not activations, not the KV cache. During inference, activations flow through in FP16/BF16, and the KV cache continues to grow with context length and batch size. AWQ weight compression compounds with KV cache quantization for further savings at serving time.

AWQ requires calibration data—128 sequences is the standard default. The AWQ paper uses samples from the Pile (a diverse multi-domain corpus) rather than a single-domain source to avoid overfitting. Our experiments use WikiText-2 for consistency with the GPTQ experiments in Section 7.3; in

production, a diverse corpus is preferable. Because AWQ only measures average activation magnitudes per channel—a low-dimensional statistic that stabilizes quickly—it is relatively robust to corpus choice.

AWQ's most significant practical advantage over GPTQ is generalization. GPTQ's layer-by-layer error reconstruction can overfit to calibration activation patterns, distorting behavior on out-of-distribution inputs. The paper quantifies this: when calibration and evaluation come from different domains (PubMed vs. Enron emails, OPT-6.7B at INT3-g128), GPTQ's perplexity degrades by 2.3–4.9 points while AWQ degrades by only 0.5–0.6. This robustness is why AWQ has become the default for instruction-tuned chatbots and multi-modal models—applications where deployment distribution is broad and calibration data cannot cover every use case.

The main conceptual difference is the absence of error compensation. GPTQ's column-by-column correction can recover from errors that scaling alone cannot prevent. AWQ's per-channel scaling is a simpler, more global optimization that works well when the activation–weight importance structure is strong, which it is for most transformer layers, especially early ones. In late layers where activation distributions become diffuse, AWQ's advantage narrows and GPTQ's Hessian-guided approach may retain a larger edge.

In practice, both methods produce comparable quality at 4-bit with group size 128. The choice often comes down to toolchain compatibility: AWQ models are natively supported by vLLM, TGI, and llama.cpp; GPTQ by the same runtimes plus ExLlamaV2.

Reproducing these results: The companion scriptch7/ch7_awq_quantization.py generates all experiments and figures from this section.

```
# Full pipeline on Colab T4 (experiments 1-4 + flow diagram)
python ch7_awq_quantization.py --mode all --save-plots

# Production deployment + perplexity (requires pip install autoaw
python ch7_awq_quantization.py --mode deploy --save-plots
python ch7_awq_quantization.py --mode perplexity --save-plots
```

```
# CPU-only (uses OPT-125M, illustrative only)
python ch7_awq_quantization.py --mode all --save-plots \
    --model opt-125m --device cpu
```

7.5 Compress KV cache with vector quantization (TurboQuant)

LLM.int8(), GPTQ, and AWQ all leave one tensor untouched: the KV cache. That tensor is the serving multiplier — at long context, it decides how many users a GPU can carry (Table 7.1).

Standard scalar KV quantization reaches roughly 4× compression at INT4. Going below 4 bits needs a different approach, because the KV cache arrives one vector at a time during generation, with no opportunity for offline calibration.TurboQuant (Zandieh et al., ICLR 2026) is a data-oblivious method that compresses any vector on arrival by separating magnitude from direction, then applying a random rotation that makes per-coordinate scalar quantization provably near-optimal.

The random rotation trick

Think of shuffling a deck of 128 cards, each marked with a coordinate magnitude. Before shuffling, the magnitudes might be concentrated in a few positions — the same outlier-channel pattern we have been chasing all chapter. After a thorough shuffle, every position is equally likely to hold any card, and the distribution at each position becomes *predictable* regardless of the starting arrangement.

Mathematically, the "shuffle" is multiplication by a random orthogonal matrix Π. After rotating a unit vector in R^d, each coordinate follows Beta(d/2,d/2) on [−1,1] —an exact result from the geometry of the unit sphere. The rotation preserves lengths and angles: no information is lost, only redistributed. And because this distribution is the same for *any* input vector, a single pre-computed quantizer works universally.

The algorithm and implementation

TurboQuant processes each key or value vector *v* (128 dimensions from one attention head) in three steps.

Step 1: Separate magnitude from direction. Compute the L2 norm $\|v\|$ and store it in FP16. Normalize: $v_{hat}=v/\|v\|$.

Step 2: Shuffle the coordinates. Rotate: $r=v_{hat}\cdot \Pi^{T}$. Each coordinate of *r* now follows the known Beta distribution, regardless of where *v* originally pointed.

Step 3: Quantize each coordinate independently. Each rotated coordinate is a small number near zero (the Beta distribution is sharply peaked). TurboQuant rounds it to the nearest entry in a lookup table called a Lloyd-Max codebook. For 3-bit quantization, this table has $2^3=8$ entries—eight representative values spaced unevenly across [−1,1], packed densely near zero where most values land. To quantize: find which of the 8 bins the coordinate falls into (binary search) and store the 3-bit index. To dequantize: replace the index with the bin's representative value. The grid is shaped to match the Beta distribution rather than spaced uniformly. The codebook is computed once offline and reused for every vector the model produces.

This per-coordinate approach may seem too simple, yet TurboQuant achieves vector quantization distortion within 2.7× of the information-theoretic lower bound—because the rotation makes coordinates near-independent, so quantizing them separately is almost as good as quantizing them jointly.

Dequantization reverses the three steps: centroid lookup, inverse rotation by Π, rescale by norm. Listing 7.13 implements the full pipeline.

Listing 7.13 TurboQuant vector quantization (Algorithm 1)

```
class TurboQuantizer:
    def __init__(self, head_dim, codebooks, seed=42, device="cuda
        self.head_dim = head_dim
        gen = torch.Generator()
        gen.manual_seed(seed)
        G = torch.randn(head_dim, head_dim, generator=gen)
        Pi, _ = torch.linalg.qr(G)                              #A
        self.Pi = Pi.to(device)
```

```
        self.centroids = {}
        self.boundaries = {}
        for bits, cb in codebooks.items():
            self.centroids[bits] = torch.tensor(
                cb["centroids"], dtype=torch.float32, device=devi
            self.boundaries[bits] = torch.tensor(
                cb["boundaries"], dtype=torch.float32, device=dev

    def quantize(self, vectors, bits):
        shape = vectors.shape
        flat = vectors.reshape(-1, self.head_dim).float()
        norms = torch.norm(flat, dim=1, keepdim=True)        #B
        normalized = flat / torch.clamp(norms, min=1e-8)     #C
        rotated = normalized @ self.Pi.T                     #D
        interior_b = self.boundaries[bits][1:-1]
        indices = torch.searchsorted(interior_b, rotated)    #E
        return {"indices": indices, "norms": norms.half(),
                "bits": bits, "shape": shape}

    def dequantize(self, compressed):
        c = self.centroids[compressed["bits"]]
        recon_rot = c[compressed["indices"]]                 #F
        recon_norm = recon_rot @ self.Pi                     #G
        recon = recon_norm * compressed["norms"].float()     #H
        return recon.reshape(compressed["shape"])
```

The round-trip introduces only Step 3 error and minor FP16 rounding in the stored norm; the rotation itself is exact. The orthogonal matrix Π is generated once from a seeded random Gaussian via QR decomposition and shared across all layers and heads.

Lloyd-Max codebooks

The Lloyd-Max algorithm is k-means in one dimension, run on a probability distribution instead of a dataset. Start with 8 initial representative values (for 3-bit). Assign every region of the Beta distribution to its nearest representative—the bin boundaries land at the midpoints between neighbors. Then update each representative to the center of mass of its assigned region. Alternate these two steps until convergence. The result is the set of 8 values that minimizes MSE for the Beta distribution—the same objective as k-means, solved in closed form because the distribution is known analytically. For d=128, each codebook converges in seconds of CPU time, is

deterministic, and is pre-shipped with the companion repository. Our codebooks reproduce the paper's theoretical MSE bounds at all tested bit-widths (1 through 4 bits).

Memory accounting

For a single head_dim-dimensional vector at *b* bits, the total storage is *b×d+16* bits (indices plus FP16 norm). Table 7.2 shows the resulting compression for d=128.

Table 7.4 TurboQuant memory per KV vector (head_dim = 128)

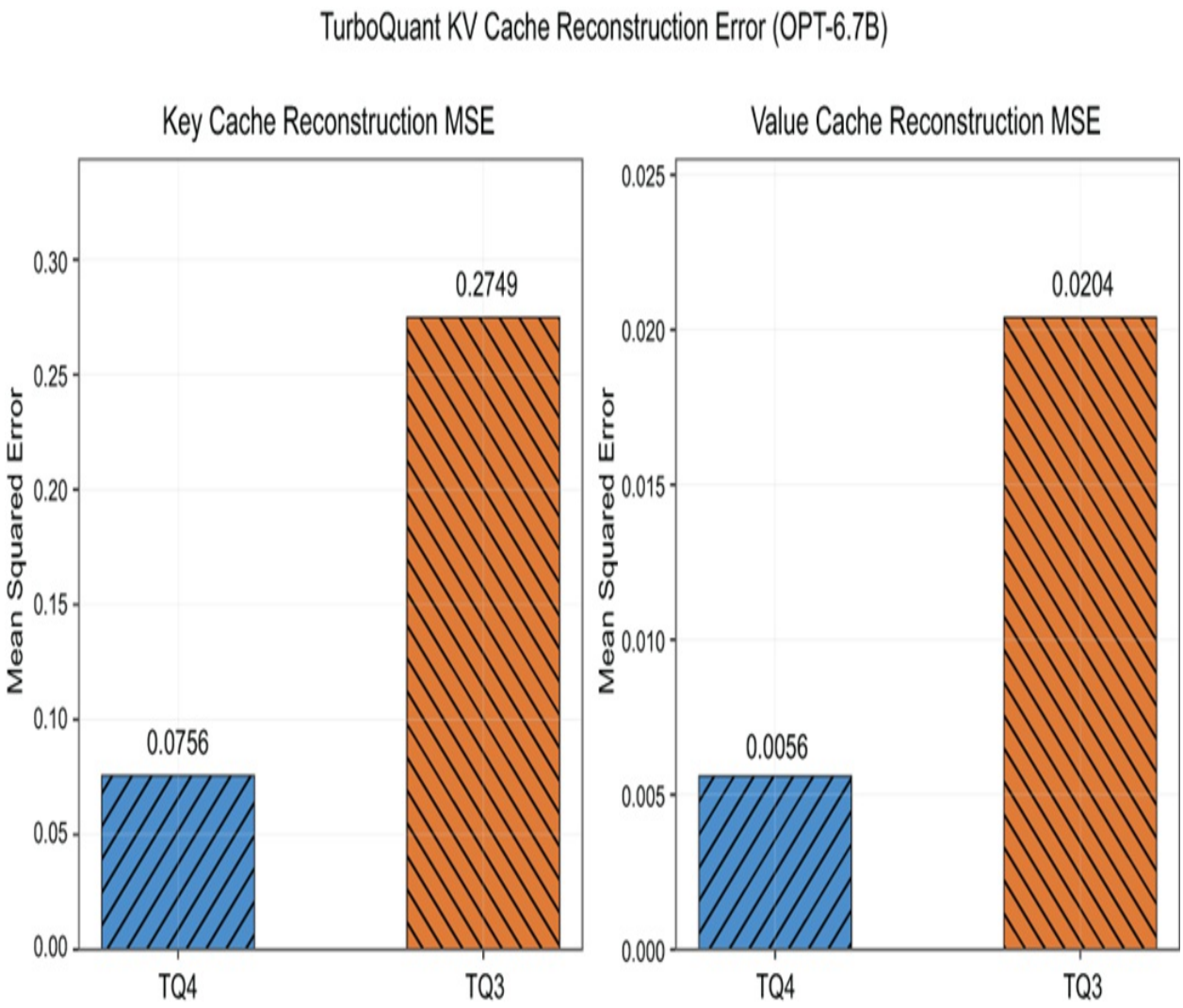
Config	Bits/elem	Bytes/token (K+V)	Compression
FP16	16.00	512	1.00×
TQ4	4.12	132	3.88×
TQ3	3.12	100	5.12×

The bits-per-element is slightly above nominal because of the 16-bit norm overhead: *(b×128+16)/128*. Note that 3-bit indices do not align to byte boundaries; production implementations use bitstream packing at finer granularity than the INT4 packing from Section 3.5.

Reconstruction MSE on real KV cache

We capture KV cache from OPT-6.7B by running 8 sequences of 512 tokens from WikiText-2 through the model and hooking every layer's key and value projections. Figure 7.18 shows the reconstruction MSE after TurboQuant round-trip.

Figure 7.18 TurboQuant reconstruction MSE on OPT-6.7B KV cache (32 layers, 8 sequences). Left: key cache. Right: value cache. TQ4 key MSE is 0.0756, value MSE is 0.0056. TQ3 roughly triples both. Values reconstruct an order of magnitude more accurately than keys, mirroring the outlier channel asymmetry from Section 3.4.



Key cache error is ~14× higher than value cache error at the same bit-width, consistent with Section 3.4's finding that keys develop concentrated outlier channels whose direction is harder to quantize. Dropping from 4 to 3 bits triples error for both keys and values, tracking the rate-distortion curve.

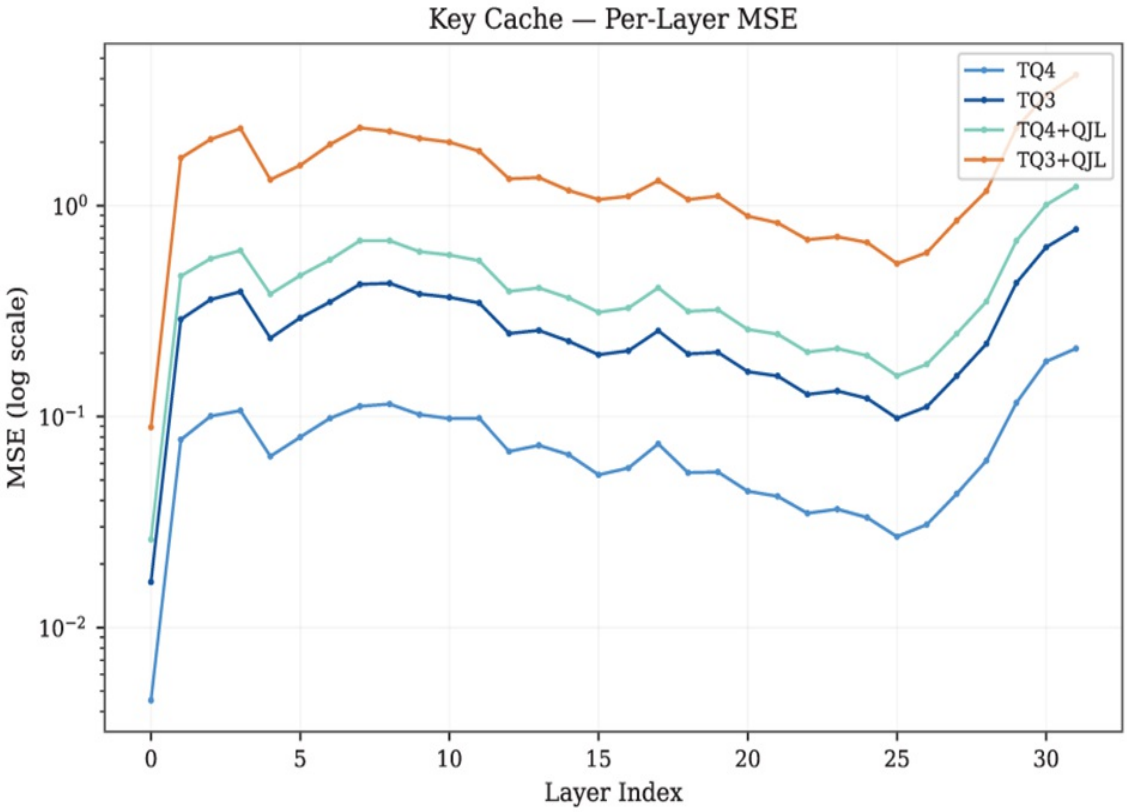
Per-layer sensitivity and the QJL question

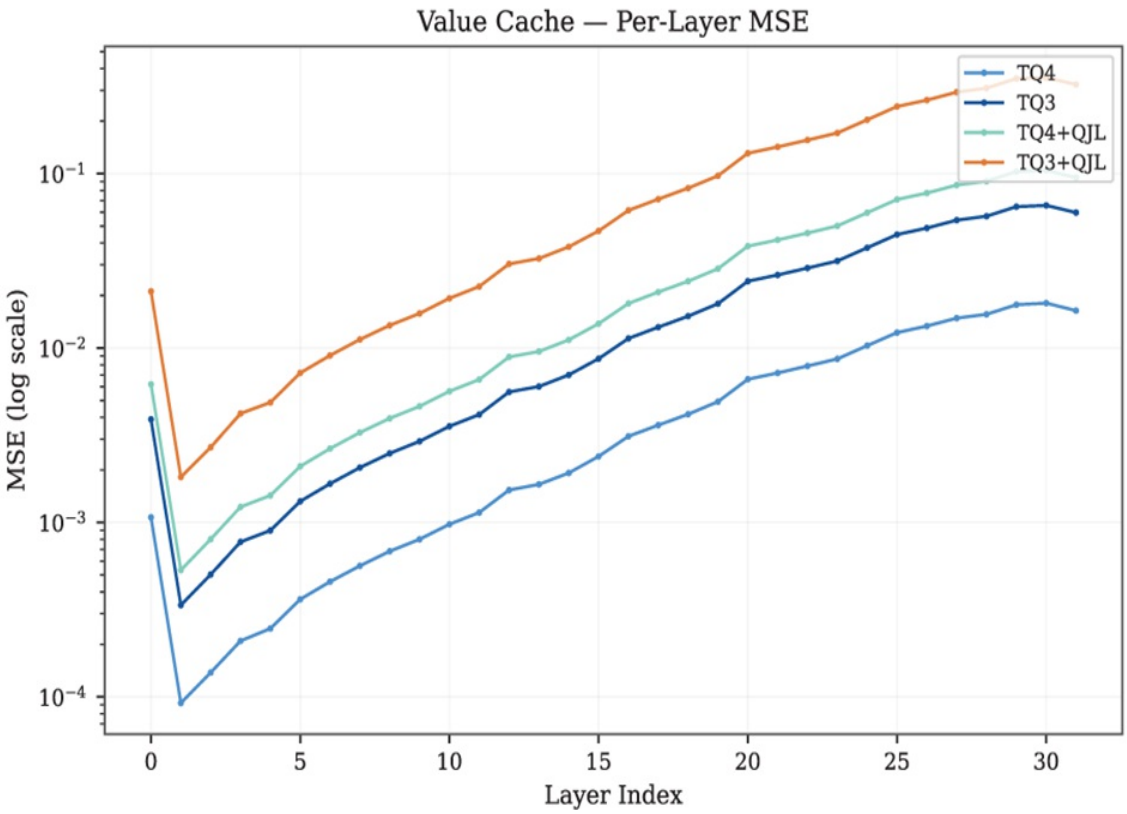
Figure 7.19 breaks down per-layer MSE and introduces a critical comparison: TurboQuant's MSE-only algorithm (Algorithm 1) versus the variant that adds a Quantized Johnson-Lindenstrauss correction (Algorithm 2, "TurboQuant-Prod").

Figure 7.19 Per-layer reconstruction MSE on OPT-6.7B with and without QJL correction. MSE-

only (solid lines) beats QJL-corrected (dashed lines) across all 32 layers, both bit-widths, and both keys and values. QJL TQ4 is worse than MSE-only TQ3 in most key cache layers. Log-scale Y-axis.

Per-Layer Sensitivity to KV Cache Quantization (OPT-6.7B)





The TurboQuant paper's Algorithm 2 reserves 1 bit for a QJL correction that produces *unbiased* inner-product estimates—useful for vector database search where ranking fidelity matters. But KV cache attention feeds inner products through softmax, which exponentiates its inputs. The exponential amplifies variance more than it benefits from unbiasedness, so the QJL variant consistently degrades perplexity. This has been confirmed across six independent implementations (Python, C, and Rust). For KV cache, always use Algorithm 1 (MSE-only).

Perplexity: FP16 vs TQ4 vs TQ3

We install forward hooks on every `k_proj` and `v_proj` in OPT-6.7B to quantize-dequantize each projection output through TurboQuant during evaluation, as Listing 7.14 shows. This is conservative—in production only cached vectors would be quantized, not the current token's projections.

Listing 7.14 Hook-based KV cache quantization for evaluation

```
def install_tq_hooks(model, tq, bits):
    hooks = []
    for layer in model.model.decoder.layers:
        for proj_name in ["k_proj", "v_proj"]:
            proj = getattr(layer.self_attn, proj_name)
            def make_hook(b):
                def hook_fn(module, inp, output):
                    flat = output.reshape(-1, tq.head_dim)
                    recon = tq.quantize_dequantize(flat, b)  #A
                    return recon.reshape(output.shape).to(output.
                return hook_fn
            hooks.append(proj.register_forward_hook(make_hook(bit
    return hooks
```

Table 7.5 shows the results on 64 sequences of length 512 from WikiText-2 test.

Table 7.5 WikiText-2 perplexity: TurboQuant KV cache (OPT-6.7B, 64 × 512 tokens)

Config	Perplexity	Bits/elem	Compression	PPL delta

FP16	16.31	16.00	1.00×	—
TQ4	16.43	4.12	3.88×	+0.7%
TQ3	17.05	3.12	5.12×	+4.5%

TQ4 adds 0.12 perplexity points (+0.7%) for 3.88× KV cache compression—within the evaluation variance observed in Sections 7.3 and 7.4. TQ3 costs 0.74 points (+4.5%) for 5.12× compression: a measurable degradation whose acceptability depends on deployment context.

TurboQuant is orthogonal to weight quantization. An OPT-6.7B deployment using AWQ 4-bit weights (3,613 MB) plus TQ4 KV cache achieves 3.5× weight compression and 3.9× KV cache compression simultaneously. At long context lengths where the KV cache dominates memory, this combination determines whether the model fits on a single GPU at all.

NOTE

The mental anchor for KV cache vector quantization: TurboQuant compresses KV vectors by exploiting high-dimensional geometry: random rotation makes coordinates follow a known distribution, enabling pre-computed optimal quantizers with no calibration. At 4 bits, 3.88× compression costs less than 1% perplexity on OPT-6.7B. Always use the MSE-only variant (Algorithm 1); the QJL correction (Algorithm 2) targets vector database search, and softmax amplifies its variance, making it consistently worse for attention. At INT4, standard scalar per-channel and per-token KV quantization is simpler and runtime-native; TurboQuant's advantage emerges at 3 bits and below, where its rotational redistribution of outlier energy is the mechanism that makes sub-4-bit KV cache compression viable.

Reproducing these results: The companion script[ch7/ch7_turboquant_kv_cache.py](ch7/ch7_turboquant_kv_cache.py) generates all experiments and figures from this section.

```
# Full pipeline on Colab T4
python ch7_turboquant_kv_cache.py --mode all --save-plots
```

```
# Codebook validation only (CPU-safe)
python ch7_turboquant_kv_cache.py --mode codebook --save-plots

# CPU-only (OPT-125M, illustrative only)
python ch7_turboquant_kv_cache.py --mode all --save-plots \
    --model opt-125m --device cpu
```

7.6 From method comparison to deployment choice

Three thresholds invalidate the standard quantization toolkit at LLM scale: activation outlier ratios jumping from 6× to 106×, outlier channels becoming coordinated across all layers (Jaccard 0.70), and those channels becoming functionally unclippable (600–1,000% perplexity degradation when zeroed).

Each of the four methods in this chapter addressed a specific facet of that problem. Table 7.6 consolidates the results.

Table 7.6 Method comparison on OPT-6.7B (WikiText-2 perplexity, T4 GPU)

	LLM.int8()	GPTQ	AWQ	TurboQuant
What it quantizes	Weights + activations	Weights only	Weights only	KV cache only
Bit-width	INT8 (mixed FP16)	INT4 (g128)	INT4 (g128)	3–4 bit
Calibration data	None	128 sequences	128 sequences	None (data-oblivious)
Quantization time	Zero (load-time)	~28 min	~32 min	Zero (online per-vector)
GPU memory	6.4 GB	3.6 GB	3.6 GB	Orthogonal (reduces KV)
Perplexity	16.31 (+0.00)	16.15 (−0.16)	15.67 (−0.64)	16.43 (+0.12, TQ4)
Compression	1.9× weights	3.5× weights	3.5× weights	3.9× KV cache (TQ4)

Core mechanism	Outlier decomposition	Hessian error compensation	Activation-aware scaling	Random rotation + codebook
Runtime support	bitsandbytes	vLLM, TGI, llama.cpp, ExLlamaV2	vLLM, TGI, llama.cpp	Custom hooks (research)
Key library	`bitsandbytes`	`gptqmodel`	`autoawq` / `llm-compressor`	Companion script

All perplexity deltas are within evaluation variance (±0.5 points at 64 × 512 tokens), so GPTQ and AWQ should be read as "no measurable degradation" rather than "better than FP16." The practical differences lie elsewhere: calibration requirements, quantization time, runtime ecosystem, and what gets compressed.

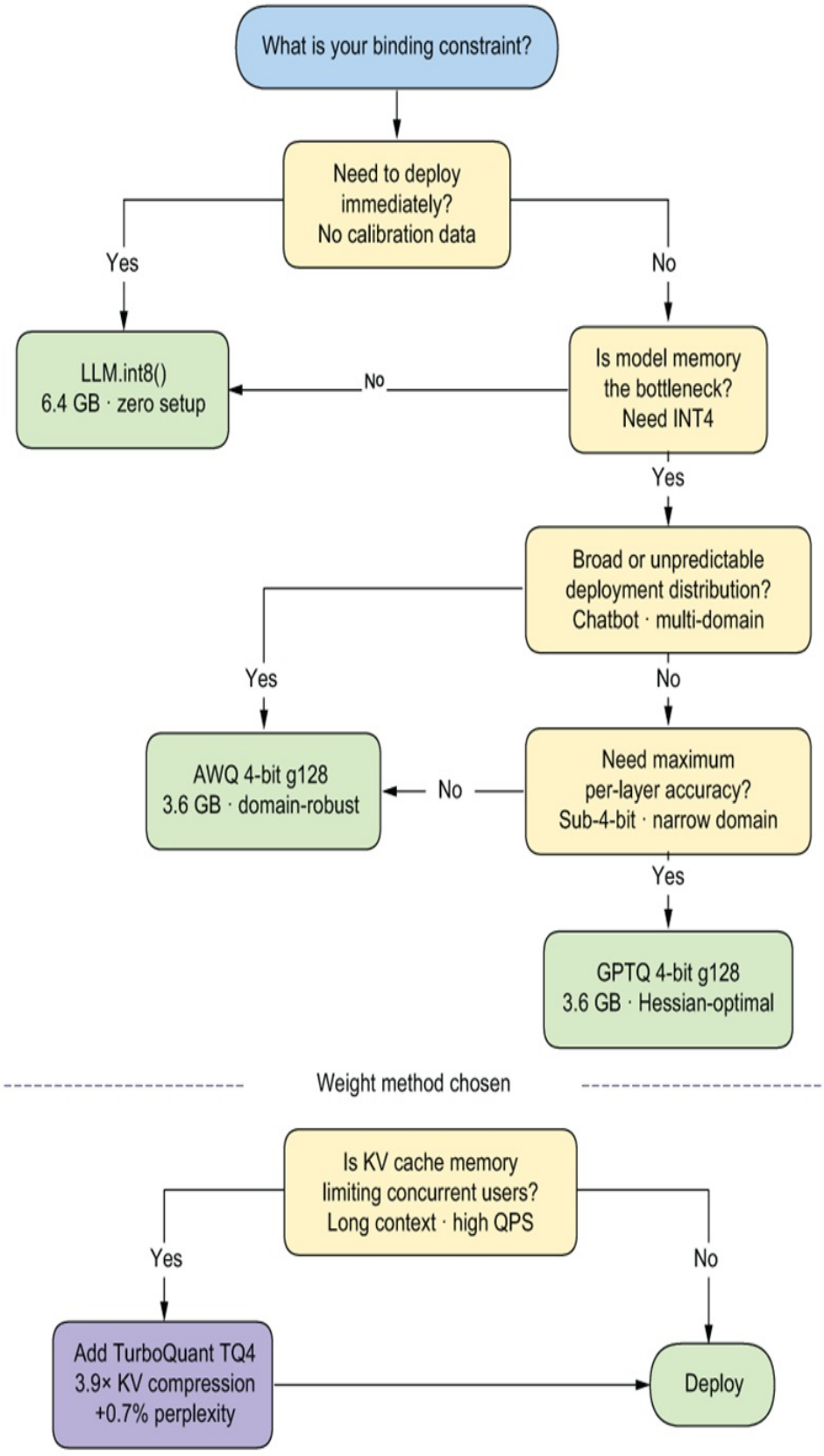
Choosing a method

The four methods are not competitors on the same axis - they target different bottlenecks and compose with each other. The decision depends on which constraint binds first: time to deployment, inference memory, serving throughput, or generalization robustness.

Figure 7.20 encodes this as a decision tree.

Figure 7.20 LLM quantization decision framework. Start from the binding deployment constraint and follow the branches. Weight methods (top) and KV cache methods (bottom) are orthogonal - combine them for maximum compression.

LLM quantization decision tree



The key branching points deserve explanation.

LLM.int8() when time-to-deploy matters most: No calibration, no offline step, no configuration beyond `load_in_8bit=True`. The trade-off is INT8 rather than INT4 — 6.4 GB instead of 3.6 GB for OPT-6.7B. This is the starting point for prototyping or any scenario where a working quantized model is needed in minutes.

AWQ when serving diverse or unpredictable workloads: Chatbots, multi-modal models, and any application where calibration data cannot cover the deployment distribution. Under domain shift (OPT-6.7B, INT3-g128), AWQ degrades by 0.5–0.6 perplexity points versus GPTQ's 2.3–4.9 — the simpler optimization generalizes better.

GPTQ when squeezing maximum quality from aggressive quantization: Hessian-guided error compensation becomes increasingly valuable below 4-bit: at 3-bit, the per-layer MSE advantage over round-to-nearest grows from 112× to 216× (Figure 7.8). Best for narrow-domain deployments where calibration data is representative — retrieval, code completion, domain-specific generation.

TurboQuant when KV cache dominates memory: Orthogonal to weight quantization — stack it with any weight method. At 128K context, a single user's FP16 KV cache can equal the model itself; TQ4 compresses it 3.9× at under 1% perplexity cost.

Combining methods in production

In practice, the typical deployment stacks one weight method with one KV cache method. The most common configurations are depicted in Table 7.7:

Table 7.7 Common deployment configurations

Scenario	Weight method	KV cache	Memory profile
Rapid prototyping	LLM.int8()	FP16	6.4 GB, zero setup

Standard serving	AWQ 4-bit g128	FP16	3.6 GB model, simple pipeline
High-concurrency serving	AWQ 4-bit g128	TQ4	3.6 GB model, 4× more users at long context
Maximum compression	GPTQ 3-bit g128	TQ3	Smallest footprint, monitor perplexity closely

7.7 Summary

- LLM quantization is a distinct problem from the toolkit in Chapters 3–6 because activation outliers at the 7B+ scale reach 100×+ the channel median, concentrate in a handful of coordinated dimensions across all layers, and cannot be clipped without collapsing perplexity.
- LLM.int8() splits each matmul into a dense INT8 path and a tiny FP16 path for the outlier dimensions, matching FP16 quality at half the memory with no calibration data — the fastest route from model to quantized deployment.
- GPTQ quantizes weight columns left-to-right and uses the input Hessian to redistribute each column's rounding error across the remaining columns, enabling 4-bit weights with no measurable quality loss on a matched calibration distribution.
- AWQ protects the ~1% of weight channels tied to high-magnitude activations by rescaling them before round-to-nearest, trading a small amount of GPTQ's optimality for much better robustness under domain shift.
- TurboQuant compresses the KV cache online with a random rotation and a fixed codebook — no calibration — and composes with any weight method, which is how you serve long-context workloads at real concurrency.

welcome

Thank you for purchasing the MEAP for *Quantization and Fast Inference*. To get the most out of this book, you'll want to be comfortable with Python and PyTorch, and have built and trained a few neural networks. Some exposure to GPU execution will help, but you don't need to be a kernel author. The book is written for ML engineers, infrastructure engineers, and applied researchers with roughly two to six years of experience. If you've shipped a model to a real system and felt the weight of its latency, memory footprint, or serving cost, you're in the right place.

Quantization has become one of the load-bearing techniques of modern AI. A 70B-parameter model in FP16 is an expensive thing to serve; the same model in 4-bit can fit on a single consumer GPU and run fast enough to hold a conversation. The gap between "this model is interesting" and "this model is deployable" is, more often than not, a quantization problem. And yet most of the material on the subject is scattered across arXiv papers, framework-specific tutorials, and blog posts that hand-wave the math and the numerical gotchas that actually bite you in production.

I wrote this book to fill that gap. Every technique in these pages is grounded in real measurements from real models running on real hardware — models you and I both use at work, not toy tensors. When you see a perplexity number, a throughput figure, or a memory breakdown, it comes out of a companion script that you can run and modify yourself. The Github repository that accompanies the book is where the prose meets the silicon: if something in the text makes you skeptical, you can re-run the experiment and check. I think this is the only honest way to write about quantization, because the field is full of claims that sound right and fall apart the moment you profile them.

The book builds up from the foundations — fixed-point arithmetic, affine mappings, the choices that govern how you map a continuous weight distribution onto a discrete grid — through the standard toolkit of post-training quantization and quantization-aware training, and into the techniques

that define the large language model era. You'll work through the formats that matter in practice (INT8, FP8, FP4, NF4, ternary), the calibration and fine-tuning recipes that make them work, the cross-framework export paths that get a quantized model out of research and into a runtime, and the deployment stacks where the speedups are actually realized. Along the way we'll confront the places where intuition misleads: why symmetric and asymmetric quantization behave differently for activations, why some "faster" formats are slower in practice, and why a model can pass every numerical check and still regress on downstream tasks.

The field moves quickly, and I'll update these chapters through the MEAP as new techniques stabilize and as your feedback shapes the book. Please post questions, corrections, and suggestions in the liveBook Discussion forum — a book like this gets sharper with every reader who pushes back on it.

—Vivek Kalyanarangan

In this book