

Memory-Efficient Integer-Only Training of Deep Neural Networks

Hazem Hesham Yousef Shalby* **Fabrizio Pittorino** **Manuel Roveri**
 Dipartimento di Elettronica, Informazione e Bioingegneria (DEIB)
 Politecnico di Milano, Milan, Italy

Abstract

Training neural networks in reduced precision can yield significant efficiency gains, yet all existing methods that scale to standard computer vision benchmarks retain floating-point in at least one stage, typically the weight update or optimizer state. We present a training algorithm that scales to modern deep neural networks, in which every tensor (weights, gradients, and activations) is stored as INT8, and every operation (forward pass, backward pass, weight update, optimizer update) is performed in integer arithmetic. Specifically, tensors are stored as signed integer mantissas paired with explicit power-of-two exponents, on which integer-only operations are performed using five custom composable building blocks (products, sums, divide-by-constant, nonlinear functions, and weight updates). This allows complete implementation of trainable integer layers, including convolutions, batch and layer normalization, softmax, and GELU. A per-group velocity mechanism with independent exponent tracking preserves gradient precision across neurons and channels with diverse magnitudes. As every tensor fits in INT8, training memory is roughly $4\times$ smaller than the FP32 baseline. We validate our training algorithm on CNNs (ResNet-18/34/50/101, MobileNetV2) and Vision Transformers (ViT-Ti/B), demonstrating for the first time that training neural networks with INT8 tensors and integer arithmetic can achieve competitive accuracy on CIFAR-10, CIFAR-100 and ImageNet at a reduced memory budget.

1 Introduction

The cost of training deep neural networks continues to grow with model size and dataset scale. A natural way to reduce this cost is to lower numerical precision, replacing 32-bit floating-point arithmetic with narrower integer operations that are cheaper in energy, area, and latency.

Mixed-precision training Micikevicius et al. [2018] showed that FP16 suffices for forward and backward propagation with minimal accuracy loss, and later methods pushed the datapath further to FP8 Micikevicius et al. [2022] and INT8 Zhu et al. [2020], Zhao et al. [2021]. However, these approaches still rely on floating-point arithmetic in at least one critical component, such as the weight update, optimizer state, or nonlinear functions.

This residual dependence on floating-point arithmetic remains a practical bottleneck for two complementary reasons. On GPUs, although floating-point computation is efficient, the persistent training state, including weights, gradients, and optimizer moments, still requires 12 bytes per parameter. As a result, model and batch size are often constrained by FP32 optimizer state rather than by the low-precision compute path itself. On integer-only platforms, including microcontrollers, fixed-function accelerators, and in-memory compute substrates, floating-point support may be absent entirely. In this setting, eliminating all floating-point operations is a fundamental prerequisite for on-device learning.

*Corresponding author: hazemhesham.shalby@polimi.it

Existing approaches that move toward fully integer training Wang et al. [2022], Hu et al. [2025] either do not include standard optimization mechanisms such as momentum and learning-rate schedules, or remain restricted to small models and tasks.

We address these challenges through a unified integer-only training framework. Our key observation is that any tensor can be represented as a signed integer mantissa together with an explicit power-of-two exponent, so arithmetic remains in the integer domain while scale is tracked exactly through exponent addition. From this representation, we derive five reusable primitives: products, sums, divide-by-constant, nonlinear functions, and weight updates. These primitives compose into complete layers with integer-only forward and backward computation. To preserve gradient precision when magnitudes vary substantially within a weight tensor, we further introduce a per-group velocity mechanism that maintains separate exponent tracking for different parameter groups, such as output channels, and enables stable integer-only SGD with momentum.

Together, these ideas define, to our knowledge, the first fully integer-only training framework spanning forward propagation, backward propagation, and parameter updates that scales to modern deep neural networks and standard computer vision benchmarks. The framework is general enough to realize standard CNN and ViT layers, including integer-only softmax, GELU, batch and layer normalization.

2 Related Work

We organize prior work by how far each approach eliminates floating-point from training; Table 1 summarizes the comparison and Table 2 lists the per-tensor storage format of each method.

Low-precision training. Mixed-precision training Micikevicius et al. [2018] showed that FP16 suffices for forward/backward passes if FP32 master weights are maintained. Subsequent methods reduced the datapath to FP8 Micikevicius et al. [2022]. Quantization-aware training methods (PACT Choi et al. [2018], LSQ Esser et al. [2020], DoReFa-Net Zhou et al. [2018], F8Net Jin et al. [2022], DQT Shalby et al. [2026]) quantize the forward pass while keeping FP32 master weights and a floating-point backward; Jacob et al. [2018] introduced an integer-arithmetic-only inference framework adopted in mobile deployment. All methods retain floating-point for the weight update.

Integer forward and backward passes. Banner et al. [2018] were the first to quantize weights, activations, and a substantial portion of the gradient stream to 8-bit on ImageNet, though the final weight-gradient accumulation remains in higher precision. Zhu et al. [2020] introduced gradient clipping strategies for full INT8 training on ImageNet-scale CNNs; Zhao et al. [2021] added per-channel scaling to close the accuracy gap. Fixed-point variants adapt bit-widths dynamically Zhang et al. [2020], while AMPA Ding et al. [2024] pushes inference below INT8 via mixed-precision allocation. All leave the optimizer in FP32.

Quasi-integer training. SWALP Yang et al. [2019] trains in 8-bit block floating-point, quantizing weights, activations, gradients, and the optimizer state via stochastic rounding; a higher-precision stochastic-weight-average running mean remains in float. WAGE Wu et al. [2018] was among the first to discretize weights, activations, gradients, and errors to low-bit integers simultaneously, but replace batch normalization layers with constant scaling to avoid floating-point operations. NITI Wang et al. [2022] performs all three stages (forward, backward, weight update) in integer arithmetic with a single per-layer block exponent, but this shared exponent forces small-gradient channels to lose their effective integer range once the scale adapts to large-gradient channels; the resulting sub-less-significant-bit (sub-LSB) vanishing is the failure mode that our per-group velocity (Section 3.3.1) addresses. NITI also lacks momentum and LR scheduling, and in practice scales only to AlexNet. Ghaffari et al. [2022] propose INT8 mantissas with an integer weight update, but their reference implementation relies on additional operations to switch back-and-forth from the floating point representation to the integer one, and falls back to float softmax in the ViT attention block. Tin-Tin Hu et al. [2025] targets microcontrollers but has been applied only to tiny sensor tasks.

Integer-only nonlinear functions. On the inference side, I-BERT Kim et al. [2021] demonstrated that polynomial approximations can replace all nonlinear functions in BERT with integer-only arithmetic, and I-ViT Li and Gu [2023] extended this to Vision Transformers with integer-only

Table 1: Comparison of low-precision training methods. ✓ = fully integer / supported; ✗ = floating-point / not supported; ~ = partially integer (some float remains). **CNN @ INet** means ResNet-18-scale or CNNs trained/evaluated on ImageNet; **ViT** means training or fine-tuning of a Vision Transformer.

Category	Method	<i>Fwd/Bwd int</i>	<i>Wt update int</i>	<i>Wt storage int</i>	<i>Momentum + LR</i>	<i>CNN @ INet</i>	<i>ViT</i>
<i>Low-precision training</i>	Mixed-Prec. Micikevicius et al. [2018]	✗	✗	✗	✓	✓	✓
	FP8 Micikevicius et al. [2022]	✗	✗	✗	✓	✓	✓
	F8Net Jin et al. [2022]	✗	✗	✗	✓	✓	✗
	DQT Shalby et al. [2026]	✗	✗	✗	✓	✓	✗
<i>Integer fwd/bwd passes</i>	Banner et al. [2018]	✓	✗	✗	✓	✓	✗
	Zhu et al. [2020]	✓	✗	✗	✓	✓	✗
	Zhao et al. [2021]	✓	✗	✗	✓	✓	✗
	Zhang et al. [2020]	✓	✗	✗	✓	✓	✗
	AMPA Ding et al. [2024]	✓	✗	✗	✓	✓	✗
<i>Quasi-integer training</i>	WAGE Wu et al. [2018]	✓	~	~	~	✗	✗
	SWALP Yang et al. [2019]	~	✓	~	✓	✓	✗
	NITI Wang et al. [2022]	✓	✓	✓	✗	✗	✗
	Ghaffari et al. [2022]	✓	~	✗	✓	✓	~
	Tin-Tin Hu et al. [2025]	✓	✓	✓	✗	✗	✗
<i>Integer-only</i>	Ours	✓	✓	✓	✓	✓	✓

Table 2: Per-tensor storage format and arithmetic precision.

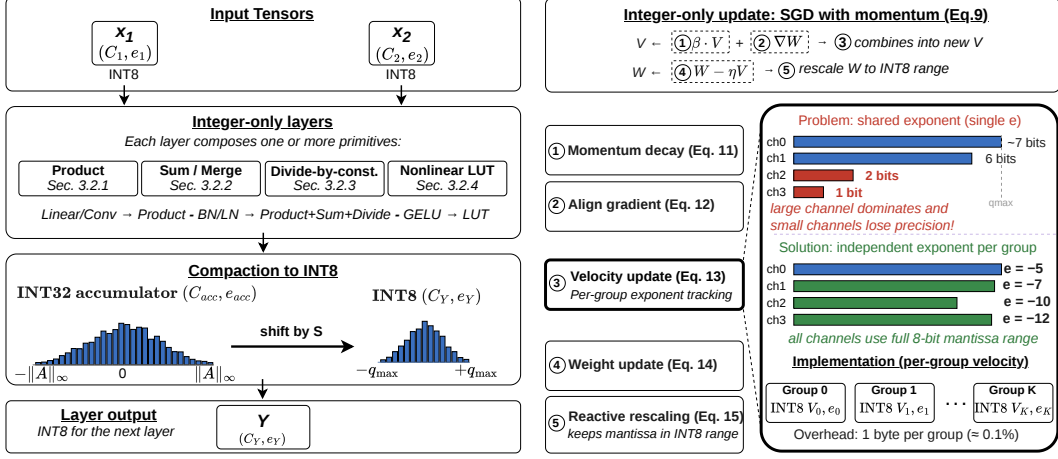
Method	Weights	Gradient	Velocity	Activations
FP32	FP32	FP32	FP32	FP32
FP16 Micikevicius et al. [2018]	FP32	FP32	FP32	FP16
FP8 NVIDIA Micikevicius et al. [2022]	FP32	FP32	FP32	FP16
FP8 IBM / HFP8 Sun et al. [2019]	FP16	FP16	FP16	FP8
INT8 fwd/bwd Jin et al. [2022]	FP32	FP32	FP32	FP32
Ghaffari et al. [2022]	FP32	FP32	INT16	FP32
Ours	INT8	INT8	INT8	INT8

Softmax, GELU, and LayerNorm. We follow a different approach, by encoding all non-linear functions in look-up tables (LUTs), without significant memory overhead (see Section 3.2.4).

3 Method

In our framework, all persistent tensors (weights, gradients, velocity, saved activations) are stored as INT8 mantissas with power-of-two exponents (Section 3.1). Arithmetic operations (inner products, sums) accumulate in transient INT32 accumulators and are immediately compacted back to INT8 by applying a tensor range dependent shift. Whenever this shift moves bits to the right, we use stochastic rounding so that the compacted value remains unbiased while still using the available integer range effectively (Figure 1a). This representation lets standard layer operations such as multiplication, sum, division, and nonlinear transformations be implemented directly on mantissas, while the corresponding exponents are updated separately through addition (Section 3.2).

For weight updates, our framework combines two mechanisms to maintain numerical efficiency and precision. First, *per-group velocity* assigns a separate exponent to each output neuron or channel, which improves precision when gradient magnitudes differ across channels (Figure 1b and Section 3.3). Second, *reactive rescaling* adjusts the weight exponent after every update step so that the



(a) Integer-only layers. Each layer composes one or more integer primitives (product, sum/merge, divide-by-constant, nonlinear LUT) on INT8 inputs into an INT32 accumulator, then compacts back to INT8 via a single shift S with eventual stochastic rounding.

(b) Integer-only weight update (Equation (9)). Five integer steps replace the float SGD-with-momentum step. Per-group velocity exponents let every channel use the full INT8 range at $< 0.1\%$ overhead (green), while a single exponent would let the largest-gradient channel dominate (red).

Figure 1: **Method overview.** All training tensors are integer (C, e) pairs (see Eq. (1)) with no floating-point storage or arithmetic at any stage. (a) Forward and backward of every layer. (b) Integer-only SGD with momentum and per-group velocity.

mantissa stays well matched to the available intB range, avoiding both overflow and under-utilization. Figure 1 summarizes the full framework.

3.1 Mantissa-exponent tensor representation

For a bit-width B we define $q_{\max} = 2^{B-1} - 1$ and store every tensor as a pair (C, e) , where $C \in [-q_{\max}, q_{\max}]^{\text{shape}}$ is a signed intB mantissa and $e \in \mathbb{Z}$ is a shared exponent ($q_{\max} = 127$ for INT8). The real-valued tensor that this represents is

$$C \cdot 2^e. \quad (1)$$

Because the exponent is a power of two, multiplying or dividing by 2^p with $p \in \mathbb{N}$ is an exact integer shift on e , so no rounding is needed by scale changes that happen to be powers of two.

Compaction. Each layer accumulates its output in an INT32 accumulator A and then stores it back in intB using a tensor range dependent shift S :

$$C_{\text{out}} = \text{CLIP}(\text{SHIFT_ROUND}(A, S), -q_{\max}, q_{\max}), \quad e_{\text{out}} = e_{\text{in}} + S. \quad (2)$$

The shift is chosen so that the stored mantissa uses the available INT8 range as fully as possible without exceeding it. Let $m = \|A\|_{\infty}$ be the maximum magnitude in A . The shift is calculated as

$$S = \lceil \log_2(m/q_{\max}) \rceil, \quad (3)$$

positive when A must be contracted and negative when it can be expanded. In practice we evaluate S as $\hat{S} = \text{bit_length}(m) - \text{bit_length}(q_{\max}) + 1$, where $\text{bit_length}(\cdot)$ is a count-leading-zeros instruction and $\text{bit_length}(q_{\max})$ is a compile-time constant.

Stochastic rounding. Right shifts are the only lossy step in the entire framework. To prevent small but nonzero gradient updates from systematically rounding to zero (which would stall learning), every right shift during training uses stochastic rounding²:

$$\text{SHIFT_ROUND}(x, S) = \left\lfloor \frac{x + U}{2^S} \right\rfloor, \quad U \sim \text{Unif}\{0, 1, \dots, 2^S - 1\}, \quad (4)$$

²Training ResNet-18 on CIFAR-10 with our algorithm and deterministic rounding achieves only 89.3%, a 5.4% gap compared to the stochastic rounding version (94.7%), confirming the advantage of stochastic rounding into the integer-only training pipeline.

where $\lfloor \cdot \rfloor$ is the floor (rounding toward $-\infty$). The estimator is unbiased for all signed x Ghaffari et al. [2022].

3.2 Layer building blocks

Given the representation above, the forward and backward passes of all standard layers reduce to four primitives on integer mantissas, with exponents propagated by addition: products, sums, divide-by-constant, and nonlinear lookup tables. The first three (Sections 3.2.1 to 3.2.3) are arithmetic; the fourth (Section 3.2.4) tabulates each elementwise nonlinearity once at initialization, since under the (C, e) representation each input (x_{int}, e_x) takes only $2^B N_e$ distinct values, with $N_e \approx 30$ exponents reached in practice. Section B works through how these primitives compose into the layers of our models, namely convolutions, linear layers, BN/LN, the ViT attention block, and the output head.

3.2.1 Products

Let (X, e_X) and (W, e_W) be an activation and weight tensor. An inner-dimension- K dot product is performed on the mantissas in an INT32 accumulator, with the exponent obtained by addition:

$$y_{\text{acc}} = \sum_{i=1}^K X_i W_i, \quad e_{\text{acc}} = e_X + e_W, \quad (5)$$

so that $(y_{\text{acc}}, e_{\text{acc}})$ represents the real value $y_{\text{real}} = y_{\text{acc}} \cdot 2^{e_{\text{acc}}}$. Compacting via Equation (3) yields (C_Y, e_Y) with $e_Y = e_{\text{acc}} + S$. The backward products $\nabla X_{\text{acc}} = GW$ and $\nabla W_{\text{acc}} = G^\top X$ are computed in the same way.

3.2.2 Sums

Residual connections and gradient stream merges add two tensors (C_1, e_1) and (C_2, e_2) , which requires aligning them to a common exponent. Assume $e_1 \geq e_2$ and let $\Delta e = e_1 - e_2$. When $\Delta e \leq \Delta e_{\text{max}}$ we left-shift C_1 by Δe (exact, no rounding) and set $e_{\text{common}} = e_2$; otherwise we right-shift C_2 by Δe with stochastic rounding and set $e_{\text{common}} = e_1$. The cap Δe_{max} prevents the left shift from overflowing the wide accumulator. The aligned mantissas (C'_1, C'_2) are summed and accumulated in INT32 and compacted back to intB:

$$C_{\text{out}} = \text{CLIP}(\text{SHIFT_ROUND}(C'_1 + C'_2, S), -q_{\text{max}}, q_{\text{max}}), \quad e_{\text{out}} = e_{\text{common}} + S. \quad (6)$$

3.2.3 Divide-by-constant

Average pooling and normalization statistics divide a tensor (C, e) by a positive integer constant D . When $D = 2^k$ with $k \in \mathbb{N}$ is a power of two, the division is absorbed into the exponent at no cost:

$$(C, e) / 2^k = (C, e - k). \quad (7)$$

For general D , we precompute the integer reciprocal $R[D] = \lfloor 2^{S_R} / D \rfloor$, so that $R[D] \cdot D \approx 2^{S_R}$ and the division becomes a multiply followed by a shift by S_R :

$$(C, e) / D \approx (\text{SHIFT_ROUND}(C \cdot R[D], S_R), e). \quad (8)$$

3.2.4 Nonlinear lookup tables

Each nonlinearity is a finite map tabulated once at initialization and evaluated at runtime by an indexed lookup plus a handful of integer ops. The tables are shared across all instances of a given operator and require no polynomial approximation or floating-point fallback.

Softmax. Softmax turns logits into a probability distribution via $\text{softmax}(x_i) = \exp(x_i) / \sum_j \exp(x_j)$, which we realize in three integer steps: (i) calculate $x'_i = x_i - \max_i(x_i)$ so that $\exp(x'_i) \in (0, 1] \ \forall \ i$; (ii) read each $\exp(x'_i)$ from an INT8 LUT; (iii) normalize by an integer reciprocal of the row sum (applied as in Eq. (8)). The probabilities are stochastically requantized to unsigned INT8 and cached for the backward pass, which takes the upstream gradient ∂y from the next layer and evaluates the softmax Jacobian $\partial x_i = p_i(\partial y_i - \sum_j \partial y_j p_j)$ on the cached probabilities p_i . When followed by cross-entropy, the loss-gradient is $\partial \mathcal{L} / \partial X_L = \frac{1}{N}(\mathbf{p} - \mathbf{1}_y)$, so backpropagation starts from an integer tensor with no need of float fallback for the loss.

GELU. We tabulate $\text{GELU}(x) = x \Phi(x)$ (where Φ is the cumulative distribution function for the Gaussian distribution) and its derivative into a pair of INT8 LUTs indexed by (x_{int}, e_x) . Forward is one lookup; backward is one lookup, one integer multiply, and one shift.

3.3 Integer-only weight update with scale alignment

Our target update is SGD with momentum applied to each weight tensor W , using learning rate η , momentum coefficient $\beta \in [0, 1)$, a persistent *velocity* (momentum buffer) V of the same shape as W , and the loss gradient ∇W produced by backpropagation. Each step applies

$$V \leftarrow \beta V + \nabla W, \quad W \leftarrow W - \eta V. \quad (9)$$

Realizing Equation (9) entirely on integer mantissa-exponent pairs exposes two scale-alignment problems that are much less pronounced in the forward and backward passes. First, gradient magnitudes can vary substantially across output channels within the same weight tensor; with a single shared exponent, large-gradient channels dominate the available dynamic range and updates for smaller channels are heavily quantized or vanish entirely. Second, after many updates the integer weight mantissa drifts: $\|W\|_\infty$ may overflow INT8, or shrink so that most of the available range goes unused, in either case losing precision at the next subtraction. We address the first with per-group velocity (Section 3.3.1) and the second with reactive rescaling (Section 3.3.3).

3.3.1 Per-group velocity

Rather than storing the velocity V as one tensor with a single shared exponent, we partition it along the output-neurons/channels axis of the weights W into N_g groups, each with its own exponent:

$$(V_k, e_{V,k}), \quad V_k \in [-q_{\max}, q_{\max}]^{\text{shape}(W_k)}, \quad k = 1, \dots, N_g, \quad (10)$$

where W_k denotes the corresponding slice of the weight tensor; the weights themselves keep a single shared exponent e_W , since their per-tensor scale changes only slowly across steps (Section 3.3.3). Each slice ∇W_k (along the same axis as the velocity) of the gradient ∇W is compacted via Eq. (2) with its own shift S_k read from $\|\nabla W_k\|_\infty$, yielding $(G_k, e_{G,k})$ with $e_{G,k} = e_{\nabla W} + S_k$. Both V_k and G_k therefore use the full INT8 range regardless of the scales of other slices. The storage overhead is one exponent per velocity slice, below 0.1% for a typical fully connected or convolution layer; all velocity mantissas are INT8 in our experiments.

3.3.2 Update algorithm

We encode the scalar hyperparameters as integer (num, shift) pairs: $\beta \approx \beta_{\text{num}} 2^{-\beta_{\text{shift}}}$ (e.g., $\beta=0.9 \approx 230/2^8$) and $\eta \approx \eta_{\text{num}} 2^{-\eta_{\text{shift}}}$, precomputing a sequence $(\eta_{\text{num}}^{(t)}, \eta_{\text{shift}}^{(t)})$ per iteration t to realize cosine, step, or warmup schedules (Section D). All arithmetic below is applied independently per group k .

We first apply momentum decay to the previous velocity:

$$D_k = \text{SHIFT_ROUND}(\beta_{\text{num}} \cdot V_k, \beta_{\text{shift}}), \quad e_{D,k} = e_{V,k}. \quad (11)$$

We then align the grouped gradient $(G_k, e_{G,k})$ to the velocity exponent $e_{V,k}$. With $\delta_k = e_{G,k} - e_{V,k}$, the aligned gradient is

$$\tilde{G}_k = \text{SHIFT_ROUND}(G_k, -\delta_k). \quad (12)$$

The new velocity is obtained by accumulation followed by compaction back to INT8:

$$V_k \leftarrow \text{CLIP}(\text{SHIFT_ROUND}(D_k + \tilde{G}_k, S'_k), -q_{\max}, q_{\max}), \quad e_{V,k} \leftarrow e_{V,k} + S'_k, \quad (13)$$

with S'_k chosen by Equation (3) from $\|D_k + \tilde{G}_k\|_\infty$. Finally, we form the update direction $\eta_{\text{num}} \cdot V_k$ at exponent $-\eta_{\text{shift}} + e_{V,k}$, align it to the weight exponent e_W (as in Equation (12)), and subtract from the current weights in a single shift-round:

$$W_k \leftarrow W_k - \text{SHIFT_ROUND}(\eta_{\text{num}} \cdot V_k, e_W + \eta_{\text{shift}} - e_{V,k}). \quad (14)$$

Weight decay is handled the same way as the momentum coefficient: λ is stored as $(\lambda_{\text{num}}, \lambda_{\text{shift}})$ and λW_k is added to G_k before the velocity update (see Section D for details).

3.3.3 Reactive rescaling

After the groupwise updates have been applied, the integer weight tensor may either exceed the INT8 representable range or use only a small fraction of it. Letting $m = \|W\|_\infty$, a single signed shift restores the mantissa to the band $[q_{\max}/4, q_{\max}]$:

$$\begin{aligned} W &\leftarrow \text{SHIFT_ROUND}(W, S), \\ e_W &\leftarrow e_W + S, \end{aligned} \quad S = \begin{cases} \lceil \log_2(m/q_{\max}) \rceil, & m > q_{\max}, \\ -\lfloor \log_2(q_{\max}/(4m)) \rfloor, & 0 < m < q_{\max}/4, \\ 0, & \text{otherwise.} \end{cases} \quad (15)$$

The first case ($S > 0$) rounds the overflow mantissa down by S bits, keeping the represented real value within one LSB; the second case ($S \leq 0$) is an exact left shift.

4 Experiments

Setup. We evaluate three architecture classes across three datasets: ResNet He et al. [2016] on CIFAR-10, CIFAR-100 Krizhevsky [2009], and ImageNet Deng et al. [2009]; MobileNetV2 Sandler et al. [2018] on CIFAR-10; ViT-Ti from scratch on CIFAR-10; and ViT-B Dosovitskiy et al. [2021] fine-tuned from a pretrained checkpoint on CIFAR-10. All runs use SGD with momentum $\beta=0.9$ (encoded as $(\beta_{\text{num}}, \beta_{\text{shift}})=(230, 8)$ per Section 3.3.2) and a cosine LR schedule with linear warmup, precomputed as $(\eta_{\text{num}}, \eta_{\text{shift}})$ pairs. The default configuration uses INT8 for activations, weights, gradients, and velocity, with INT32 accumulators. Full per-model hyperparameters, augmentation, input quantization, and the integer LR schedule are listed in Section D.

Main results. Table 3 summarizes the comparison. Ours is the only method that holds every persistent tensor in the training pipeline at INT8 (weights, gradients, velocity, and saved activations), yielding a $4\times$ reduction in training memory across every configuration while staying within a few percent of FP32 accuracy and within 0.1–0.7% of Ghaffari et al. [2022], despite eliminating their FP32 master weights, FP32 optimizer state, and float softmax fallback. The memory gap to other low-precision methods is structural: training memory is dominated by saved-for-backward activations Chen et al. [2016] (80–99% of the FP32 footprint across our configurations, per-config breakdown in Section C), and prior INT8 forward/backward methods leave activations in FP32 between layers. The Pareto front corresponding to each method in Table 3 is reported in Figure 2. Our algorithm is particularly advantageous at low memory budgets: e.g., on ImageNet it reaches 76.4% at 5.20 GB memory on ResNet-50, matching the FP32 ResNet-18 (4.61 GB, 70.6%) memory footprint within 13% with an increase of +5.8% in accuracy.

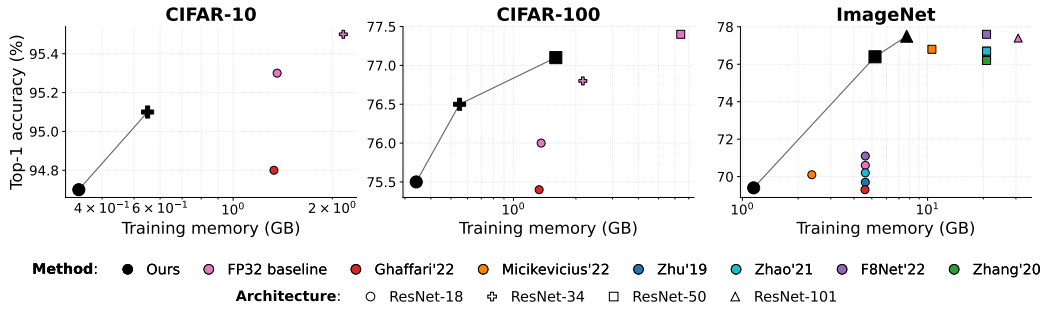


Figure 2: Accuracy vs training memory on CIFAR-10, CIFAR-100 and ImageNet. Each point is one (method, model) entry from Table 3.

CNN results. On ResNet-18 we trail FP32 by 0.6% on CIFAR-10 (94.7 vs. 95.3), 0.5% on CIFAR-100 (75.5 vs. 76.0), and 1.2% on ImageNet (69.4 vs. 70.6), at roughly $4\times$ smaller training-memory footprint than the baselines. On ResNet-50/ImageNet our algorithm reaches 76.4%, a 0.3% gap from FP32 (76.7%) and within 1.2% of the best INT8 forward/backward result (Jin et al. [2022], 77.6%), while running at 5.20 vs. 20.82 GB (same $4\times$ memory reduction). MobileNetV2/CIFAR-10 widens to -3.2% (91.5 vs. 94.7), reflecting the architecture known sensitivity to quantization. Two factor

compound: depthwise separable convolutions decompose each spatial filter into per-channel kernels with very few weights, making each channel gradient signal small and quantization-sensitive, even with our per-output-channel velocity (which contributes +2.2%, see Table 5); and the inverted-residual blocks introduce a high density of sequential requantization points across depth. To our knowledge this is the first fully integer training of ResNet and MobileNet architectures.

Table 3: Test accuracy (%) and training memory (GB) on CIFAR-10, CIFAR-100 and ImageNet. Training details and hyperparameters are reported in Section D, while the memory computation is detailed in Section C.

Dataset	Model	Method	Acc. (%)	Training mem. (GB)	Precision
CIFAR-10	ResNet-18	FP32 Ghaffari et al. [2022]	95.3	1.36	FP32
		Ghaffari et al. [2022]	94.8	1.33	INT8/16 + FP32
		Ours	94.7	0.34	INT8
	ResNet-34	FP32 Mert Bülent Sanyıldız [2018]	95.5	2.16	FP32
		Ours	95.1	0.55	INT8
	MobileNetV2	FP32 Zhao et al. [2021]	94.7	1.82	FP32
		Zhu et al. [2020]	93.3	1.82	INT8 + FP32
		Zhao et al. [2021]	94.3	1.82	INT8 + FP32
		Ours	91.5	0.45	INT8
	ViT-Ti	FP32 Aleptonic [2025]	90.9	7.60	FP32
		Ours	88.0	1.90	INT8
CIFAR-100	ResNet-18	FP32 Ghaffari et al. [2022]	76.0	1.36	FP32
		Ghaffari et al. [2022]	75.4	1.33	INT8/16 + FP32
		Ours	75.5	0.34	INT8
	ResNet-34	FP32 weiaicunzai [2017]	76.8	2.16	FP32
		Ours	76.5	0.55	INT8
	ResNet-50	FP32 weiaicunzai [2017]	77.4	6.40	FP32
		Ours	77.1	1.60	INT8
	ResNet-18	FP32 Micikevicius et al. [2022]	70.6	4.61	FP32
		Micikevicius et al. [2022]	70.1	2.37	FP8 + FP32
		Zhu et al. [2020]	69.7	4.61	INT8 + FP32
		Zhao et al. [2021]	70.2	4.61	INT8 + FP32
		Jin et al. [2022]	71.1	4.61	INT8 + FP32
		Ghaffari et al. [2022]	69.3	4.59	INT8/16 + FP32
		Ours	69.4	1.15	INT8
ImageNet	ResNet-50	FP32 Micikevicius et al. [2022]	76.7	20.82	FP32
		Micikevicius et al. [2022]	76.8	10.55	FP8 + FP32
		Zhu et al. [2020]	76.3	20.82	INT8 + FP32
		Zhang et al. [2020]	76.2	20.82	INT8 + FP32
		Zhao et al. [2021]	76.7	20.82	INT8 + FP32
		Jin et al. [2022]	77.6	20.82	INT8 + FP32
		Ours	76.4	5.20	INT8
	ResNet-101	FP32 TorchVision [2016]	77.4	30.84	FP32
		Ours	77.5	7.71	INT8

Vision Transformers. ViT-Ti from scratch on CIFAR-10 reaches 88.0%, a -2.9% gap from FP32 (90.9%). ViT-B fine-tuning on CIFAR-10 reaches 96.1% in 50 epochs, a -3.0% gap from FP32 (99.1%). To our knowledge, this is the first fully integer-only training or fine-tuning of a Vision Transformer: every layer (attention softmax, $1/\sqrt{d_h}$ scaling, LayerNorm, and GELU) runs on integer mantissa-exponent pairs with no floating-point fallback at any stage, (see Table 11 for the accuracy impact of quantizing each layer). The closest prior point, Ghaffari et al. [2022], retains FP32 master weights, an FP32 optimizer, and a float softmax in attention. ViTs are typically trained

with Adam/AdamW, while our framework supports only SGD+momentum at the moment; extending the integer pipeline to adaptive optimizers could improve performance and is a natural next step, since their moment updates require integer square roots and divisions that complicate the scale bookkeeping.

Lower-bit regimes. Table 4 sweeps the bit-width B below INT8 on ResNet-18/CIFAR-10 on our algorithm and on the algorithm from Ghaffari et al. [2022]. Both methods retain their default optimizer state while weights and activations drops to B bits: our velocity stays at INT8, while Ghaffari et al. [2022] master weights stay at FP32 and their velocity at INT16. Notably, our algorithm retains 70.5% at $B=4$ where Ghaffari et al. [2022] diverges. Memory scales with B for ours (0.34→0.17 GB), while Ghaffari et al. [2022] retains the persistent FP32 state.

Table 4: Accuracy (%) and training memory (GB) on ResNet-18 / CIFAR-10 (batch 256) across bit-widths $B \in \{8, 7, 6, 5, 4\}$.

Method	Metric	INT8	INT7	INT6	INT5	INT4	Precision ($B \in \{4, \dots, 8\}$)
Ghaffari et al. [2022]	Acc (%)	94.8	94.7	94.47	88.5	Diverge	intB + FP32, velocity INT16
	Mem (GB)	1.33	1.33	1.33	1.33	1.33	
Ours	Acc (%)	94.7	94.1	91.8	83.3	70.5	intB, velocity INT8
	Mem (GB)	0.34	0.30	0.26	0.22	0.17	

Per-group velocity ablation. Table 5 isolates the contribution of per-group velocity (Section 3.3.1) on CIFAR-10. Disabling it costs 0.9% on ResNet-18, 2.2% on MobileNetV2, and 2.7% on ViT-Ti.

Table 5: Effect of per-group velocity on test accuracy (%) and exponent memory.

Dataset	Model	FP32	w/o per-group velocity		w/ per-group velocity	
		(Reference)	Acc.	Exp. mem.	Acc.	Exp. mem.
CIFAR-10	ResNet-18	95.3	93.8 (−1.5)	0.02 KB	94.7 (−0.6)	4.7 KB
	MobileNetV2	94.7	89.3 (−5.4)	0.05 KB	91.5 (−3.2)	16.7 KB
	ViT-Ti	90.9	85.3 (−5.6)	0.05 KB	88.0 (−2.9)	20.5 KB

5 Conclusion

We demonstrated that a full SGD-with-momentum training pipeline can be realized end to end on INT8 mantissa-exponent tensors with no floating-point fallback at any stage. The resulting algorithm is, to our knowledge, the first integer-only training framework that scales to standard computer-vision benchmarks at modern depth: it trains ResNet-18/34/50/101 and MobileNetV2 to within a few percent of FP32 across CIFAR-10, CIFAR-100, and ImageNet, fine-tunes ViT-B integer-only, and trains ViT-Ti from scratch. Methodologically, per-output-channel velocity exponents are essential for integer optimizer state. Because every persistent tensor is INT8, training memory is roughly $4\times$ smaller than FP32 across every configuration we tested. This shifts the accuracy-memory frontier: at a fixed memory budget, the integer pipeline trains substantially deeper and more accurate architectures than FP32 can admit; at a fixed architecture, it trains within a few percent of the FP32 baseline at a quarter of the memory. The same pipeline can in principle run on hardware without floating-point support (e.g. microcontrollers, in-memory compute substrates) opening on-device training of ResNet- and ViT-scale models.

Limitations and future work. Extending the framework to adaptive optimizers (an integer Adam, requiring integer square roots, integer division, and exponent bookkeeping for the bias-correction terms) is the most substantive open direction for moving to architectures that are typically trained with Adam or AdamW. Replacing power-of-two exponents with richer scaling schemes (learned scales, non-power-of-two block exponents) would use dynamic range more efficiently and likely narrow the remaining accuracy gap on MobileNetV2 and ViT. Operator fusion (e.g. reducing the

requantization points per ViT block) is a possible engineering direction for requantization-heavy architectures. Finally, bringing the framework to LLM training and parameter-efficient fine-tuning (LoRA) is a natural next step, with the same structural advantage of keeping the optimizer state at INT8 rather than FP32.

Acknowledgments and Disclosure of Funding

This work was carried out in the EssilorLuxottica Smart Eyewear Lab, a Joint Research Center between EssilorLuxottica and Politecnico di Milano.

References

- Aleptonic. Vision transformer from scratch. <https://github.com/Aleptonic/Vision-Transformer-From-Scratch>, 2025. GitHub repository.
- Ron Banner, Itay Hubara, Elad Hoffer, and Daniel Soudry. Scalable methods for 8-bit training of neural networks, 2018. URL <https://arxiv.org/abs/1805.11046>.
- Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost, 2016. URL <https://arxiv.org/abs/1604.06174>.
- Jungwook Choi, Zhuo Wang, Swagath Venkataramani, Pierce I-Jen Chuang, Vijayalakshmi Srinivasan, and Kailash Gopalakrishnan. Pact: Parameterized clipping activation for quantized neural networks, 2018. URL <https://arxiv.org/abs/1805.06085>.
- Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255, 2009. doi: 10.1109/CVPR.2009.5206848.
- Li Ding, Wen Fei, Yuyang Huang, Shuangrui Ding, Wenrui Dai, Chenglin Li, Junni Zou, and Hongkai Xiong. AMPA: Adaptive mixed precision allocation for low-bit integer training. In Ruslan Salakhutdinov, Zico Kolter, Katherine Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp, editors, *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pages 10960–10977. PMLR, 21–27 Jul 2024. URL <https://proceedings.mlr.press/v235/ding24b.html>.
- Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale, 2021. URL <https://arxiv.org/abs/2010.11929>.
- Steven K. Esser, Jeffrey L. McKinstry, Deepika Bablani, Rathinakumar Appuswamy, and Dharmendra S. Modha. Learned step size quantization. In *International Conference on Learning Representations (ICLR)*, 2020. URL <https://openreview.net/forum?id=rkg066VKDS>.
- Alireza Ghaffari, Marzieh S. Tahaei, Mohammadreza Tayaranian, Masoud Asgharian, and Vahid Partovi Nia. Is integer arithmetic enough for deep learning training? In *Proceedings of the 36th International Conference on Neural Information Processing Systems*, NIPS ’22, Red Hook, NY, USA, 2022. Curran Associates Inc. ISBN 9781713871088.
- Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016. doi: 10.1109/CVPR.2016.90.
- Yi Hu, Jinhang Zuo, Eddie Zhang, Bob Iannucci, and Carlee Joe-Wong. Tin-tin: Towards tiny learning on tiny devices with integer-based neural network training, 2025. URL <https://arxiv.org/abs/2504.09405>.
- Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.

- Qing Jin, Jian Ren, Richard Zhuang, Sumant Hanumante, Zhengang Li, Zhiyu Chen, Yanzhi Wang, Kaiyuan Yang, and Sergey Tulyakov. F8net: Fixed-point 8-bit only multiplication for network quantization, 2022. URL <https://arxiv.org/abs/2202.05239>.
- Sehoon Kim, Amir Gholami, Zhewei Yao, Michael W. Mahoney, and Kurt Keutzer. I-bert: Integer-only bert quantization. In Marina Meila and Tong Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pages 5506–5518. PMLR, 18–24 Jul 2021. URL <https://proceedings.mlr.press/v139/kim21d.html>.
- Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical report, University of Toronto, 2009.
- Zhikai Li and Qingyi Gu. I-vit: Integer-only quantization for efficient vision transformer inference, 2023. URL <https://arxiv.org/abs/2207.01405>.
- Mert Bülent Sarıyıldız. resnet-pytorch: Resnet implementation in pytorch with cifar-10 baselines. <https://github.com/mbsariyildiz/resnet-pytorch>, 2018. GitHub repository.
- Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. Mixed precision training, 2018. URL <https://arxiv.org/abs/1710.03740>.
- Paulius Micikevicius, Dusan Stolic, Neil Burgess, Marius Cornea, Pradeep Dubey, Richard Grisenthwaite, Sangwon Ha, Alexander Heinecke, Patrick Judd, John Kamalu, Naveen Mellempudi, Stuart Oberman, Mohammad Shoeybi, Michael Siu, and Hao Wu. Fp8 formats for deep learning, 2022. URL <https://arxiv.org/abs/2209.05433>.
- Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 4510–4520, 2018. doi: 10.1109/CVPR.2018.00474.
- Hazem Hesham Yousef Shalby, Fabrizio Pittorino, Francesca Palermo, Diana Trojaniello, and Manuel Roveri. Dqt: Dynamic quantization training via dequantization-free nested integer arithmetic. *Proceedings of the AAAI Conference on Artificial Intelligence*, 40(30):25252–25259, Mar. 2026. doi: 10.1609/aaai.v40i30.39717. URL <https://ojs.aaai.org/index.php/AAAI/article/view/39717>.
- Xiao Sun, Jungwook Choi, Chia-Yu Chen, Naigang Wang, Swagath Venkataramani, Vijayalakshmi Srinivasan, Xiaodong Cui, Wei Zhang, and Kailash Gopalakrishnan. Hybrid 8-bit floating point (hfp8) training and inference for deep neural networks. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems*, Red Hook, NY, USA, 2019. Curran Associates Inc.
- TorchVision. TorchVision: Pytorch’s computer vision library, pretrained models. <https://github.com/pytorch/vision>, 2016.
- Maolin Wang, Seyedramin Rasoulinezhad, Philip H. W. Leong, and Hayden K.-H. So. Niti: Training integer neural networks using integer-only arithmetic. *IEEE Transactions on Parallel and Distributed Systems*, 33(11):3249–3261, November 2022. ISSN 2161-9883. doi: 10.1109/tpds.2022.3149787. URL <http://dx.doi.org/10.1109/TPDS.2022.3149787>.
- weiaicunzai. Practice on cifar-100 with pytorch. <https://github.com/weiaicunzai/pytorch-cifar100>, 2017. GitHub repository.
- Shuang Wu, Guoqi Li, Feng Chen, and Luping Shi. Training and inference with integers in deep neural networks, 2018. URL <https://arxiv.org/abs/1802.04680>.
- Guandao Yang, Tianyi Zhang, Polina Kirichenko, Junwen Bai, Andrew Gordon Wilson, and Christopher De Sa. Swalp : Stochastic weight averaging in low-precision training, 2019. URL <https://arxiv.org/abs/1904.11943>.

Xishan Zhang, Shaoli Liu, Rui Zhang, Chang Liu, Di Huang, Shiyi Zhou, Jiaming Guo, Qi Guo, Zidong Du, Tian Zhi, and Yunji Chen. Fixed-point back-propagation training. In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2327–2335, 2020. doi: 10.1109/CVPR42600.2020.00240.

Kang Zhao, Sida Huang, Pan Pan, Yinghan Li, Yingya Zhang, Zhenyu Gu, and Yinghui Xu. Distribution adaptive int8 quantization for training cnns. *Proceedings of the AAAI Conference on Artificial Intelligence*, 35(4):3483–3491, May 2021. doi: 10.1609/aaai.v35i4.16462. URL <https://ojs.aaai.org/index.php/AAAI/article/view/16462>.

Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, and Yuheng Zou. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients, 2018. URL <https://arxiv.org/abs/1606.06160>.

Feng Zhu, Ruihao Gong, Fengwei Yu, Xianglong Liu, Yanfei Wang, Zhelong Li, Xiuqi Yang, and Junjie Yan. Towards unified int8 training for convolutional neural network. In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1966–1976, 2020. doi: 10.1109/CVPR42600.2020.00204.

A Integer primitives, shifts, and safety caps

This appendix defines the low-level integer primitives used throughout the paper: the clip and signed shift-and-round operators, the stochastic-rounding operator, and the accumulator safety caps that bound the compaction shift of Equation (3).

Clip and shift-round. The two primitives from which every rescale in the paper is built are saturation and signed shift-with-round:

$$\text{CLIP}(x, a, b) = \min(\max(x, a), b), \quad \text{SHIFT_ROUND}(x, S) = \begin{cases} \lfloor (x + b(S))/2^S \rfloor, & S > 0, \\ x \cdot 2^{-S}, & S \leq 0, \end{cases} \quad (16)$$

where $\lfloor \cdot \rfloor$ denotes the floor toward $-\infty$ and $b(S) \geq 0$ is an integer bias that selects the rounding mode. We use $b(S) = U \sim \text{Unif}\{0, \dots, 2^S - 1\}$ during training (stochastic rounding, Equation (4)) and $b(S) = 2^{S-1}$ at inference (deterministic midpoint). Left shifts ($S \leq 0$) are exact and involve no rounding choice. The compaction shift of Equation (3) uses $\lceil \log_2 m \rceil$, computed by integer bit-length (count-leading-zeros); we handle the edge case $m = \|A\|_\infty = 0$ by setting $S = 0$.

Stochastic rounding is integer-only. Stochastic rounding samples a uniform integer U per tensor element and computes

$$\text{SHIFT_ROUND}(x, S) = \lfloor (x + U)/2^S \rfloor, \quad U \sim \text{Unif}\{0, \dots, 2^S - 1\}, \quad (17)$$

so every ingredient is integer: a xorshift or PCG generator produces U in 3–5 integer ops (shifts and xors), and the addition and right shift are native integer instructions, so the same kernel runs on hardware without an FPU. The per-element PRNG cost is negligible against the K -wide inner product that produces x : K ranges from 9 for a 3×3 depthwise convolution to 4608 for a ViT-B MLP projection, so the PRNG adds well under 1% to the per-layer arithmetic. We hold one PRNG state per tensor to avoid cross-thread contention; at inference U is replaced by the deterministic midpoint 2^{S-1} , eliminating the PRNG entirely.

Accumulator safety caps. The shift chosen by Equation (3) must not cause the next reduction to overflow. Let the wide accumulator be n -bit signed with maximum magnitude $A_{\max} = 2^{n-1} - 1$. For a K -wide inner product with $|x_i|, |w_i| \leq q_{\max}$ we have $|\sum_{i=1}^K x_i w_i| \leq K q_{\max}^2$, and for an M -way sum of aligned mantissas $|\sum_i x_i| \leq M q_{\max}$. The largest admissible expansions are therefore

$$S_{\text{cap}}^{\text{prod}} = \lfloor \log_2(A_{\max}/(K q_{\max}^2)) \rfloor, \quad S_{\text{cap}}^{\text{sum}} = \lfloor \log_2(A_{\max}/(M q_{\max})) \rfloor. \quad (18)$$

If a cap is negative the reduction must be tiled, the operands pre-shifted, or a wider accumulator used. With INT32 accumulators ($A_{\max} \approx 2.1 \cdot 10^9$) and $q_{\max} = 127$ (INT8), both caps stay comfortably positive for every layer size in our models.

B Concrete layer instantiations

Each layer is built by composing the primitives of Sections 3.2 and 3.3. Table 6 maps each layer or operator used in our models to the primitives it reuses and any layer-specific lookup table; the subsections below fill in operator-specific details (LUT definitions, backward equations, and architectural constants). Residual adds and backward gradient merges reuse the sum operator directly (Equation (6)), and average pooling reuses the divide-by-constant primitive (Equations (7) and (8)).

Table 6: Layer and operator instantiations used by our framework.

Layer / operator	Primitives used	Ref.
Convolution (incl. 1×1 , depthwise)	Product	Equations (3) and (5)
Linear / fully-connected	Product	Equations (3) and (5)
Patch embedding (strided conv)	Product	Equations (3) and (5)
Residual add, gradient merge	Sum	Equation (6)
Average pool ($K \times K$)	Divide-by-constant	Equations (7) and (8)
ReLU / ReLU6	Clip	Section B.1
GELU	Product, LUT	Section B.1
Batch / Layer normalization	Product, sum, divide-by-constant	Section B.2
Softmax	Product, divide-by-constant, LUT	Section B.3
Cross-entropy loss	Softmax, divide-by-constant, LUT	Section B.3
ViT attention (Q/K/V, score, context)	Product, softmax, constant rescale	Section B.4

B.1 Activations

Our models use three pointwise activations: ReLU (ResNet-18), ReLU6 (MobileNetV2), and GELU (ViT-Ti/B). Table 7 compares their integer forward and backward; none changes the exponent e , and only GELU requires a lookup table.

Table 7: Pointwise activations used in our models. All operations are integer on the $\text{int}B$ mantissa C ; the exponent e is carried through unchanged. dY denotes the upstream gradient.

Activation	Forward (on C)	Backward (grad)	LUT
ReLU	$\max(C, 0)$	$dY \cdot \mathbf{1}[C > 0]$	—
ReLU6	$\text{CLIP}(C, 0, c)$, $c = \min(6 \gg e, q_{\max})$	$dY \cdot \mathbf{1}[0 < C < c]$	—
GELU	$\text{LUT}_{\text{fwd}}[e, C]$ (Equation (19))	$(dY \cdot \text{LUT}_{\text{bwd}}[e, C]) \gg B_G$	INT8 [30, 256]

ReLU and ReLU6 are single-clip operations with no LUT. For ReLU6, expressing the clipping threshold 6 in the mantissa domain requires the per-exponent cap $c = \min(6 \gg e, q_{\max})$ (a plain signed shift: right by e for $e > 0$, left by $|e|$ for $e \leq 0$; no rounding is needed because c is a deterministic bound, and truncation only shrinks the clip region). When $6 \gg e \geq q_{\max}$ the clip reduces to ReLU because the mantissa already saturates below 6.

GELU LUTs. Because GELU is pointwise, its entire INT8 behavior fits in a pair of INT8 tables indexed by (e_x, x_{int}) :

$$\begin{aligned} \text{LUT}_{\text{fwd}}[e_x, x] &= \text{CLIP}(\text{ROUND}(\text{GELU}(x \cdot 2^{e_x})/2^{e_x}), -q_{\max}, q_{\max}), \\ \text{LUT}_{\text{bwd}}[e_x, x] &= \text{ROUND}(\text{GELU}'(x \cdot 2^{e_x}) \cdot 2^{B_G}), \end{aligned} \quad (19)$$

with $B_G=6$ so that $\text{GELU}' \in [-0.13, 1.13]$ fits in INT8 as $[-8, 72]$. Reducing B_G from the common 8 to 6 produces no measurable change in backward accuracy versus autograd, because the downstream compaction to INT8 already limits gradient precision to the input bit-width. The backward multiply in Table 7 stays in INT32 since $|\text{LUT}_{\text{bwd}}| \leq 72$ and $|dY| \leq q_{\max}$.

Cross-bit-width LUT reuse. For $B < 8$, an $\text{int}B$ mantissa x_B at exponent e_B equals the INT8 mantissa $x_B \ll (8-B)$ at exponent $e_B - (8-B)$; a single INT8 GELU table therefore serves all bit widths by shifting the input into the INT8 domain and stochastically shifting the result back.

B.2 Batch and layer normalization

BN and LN share the same integer pipeline; only the reduction axis differs (the batch for BN, the feature dimension for LN). Given an input mantissa C_x at exponent e_x reduced over $D=m_D 2^{s_D}$ elements, the layer computes $y = \gamma(x - \mu)/\sqrt{\sigma^2 + \epsilon} + b$ in five integer stages. A single reciprocal table $R[m_D] = \lfloor 2^{S_R}/m_D \rfloor$ (we use $S_R=30$) is precomputed once and shared by every BN/LN layer at that odd-part m_D . Every step below is integer-only: no float intermediate is constructed.

(1) Mean. The elementwise sum $T_\mu = \sum_i C_{x,i}$ is the integer sum primitive (Equation (6)); because the operands already share e_x , no alignment is needed. Dividing by $D = m_D 2^{s_D}$ is a multiply-and-shift through the reciprocal table:

$$C_\mu = \text{SHIFT_ROUND}(T_\mu \cdot R[m_D], S_R + s_D), \quad e_\mu = e_x. \quad (20)$$

(2) Variance. The centered square sum $T_{\sigma^2} = \sum_i (C_{x,i} - C_\mu)^2$ reuses the product-accumulator (Equation (5)); the safety cap (Equation (18)) covers it because $|C_{x,i} - C_\mu| \leq 2q_{\max}$. One more multiply-and-shift by $R[m_D]$ yields the variance mantissa:

$$C_{\sigma^2} = \text{SHIFT_ROUND}(T_{\sigma^2} \cdot R[m_D], S_R + s_D), \quad e_{\sigma^2} = 2e_x. \quad (21)$$

(3) Integer square root and reciprocal. We take $\sigma = \lfloor \sqrt{C_{\sigma^2} + \epsilon_{\text{int}}} \rfloor$ by the standard Newton integer square-root (a fixed number of integer shifts, adds, and compares). Count-leading-zeros decomposes $\sigma = m_\sigma 2^{s_\sigma}$, and one INT32 divide per reduction row builds $\text{inv}_\sigma = \lfloor 2^{S_R}/m_\sigma \rfloor$ (the cost amortizes over the D normalizations below); $e_\sigma = e_x$.

(4) Normalize. Each centered value is multiplied by inv_σ in an INT32 accumulator and compacted back to intB with stochastic rounding (Equation (3)):

$$(C_{\hat{x},i}, e_{\hat{x}}) = \text{COMPACT}((C_{x,i} - C_\mu) \cdot \text{inv}_\sigma, -S_R - s_\sigma). \quad (22)$$

The natural exponent passed to compact cancels both the 2^{S_R} in inv_σ and the 2^{s_σ} in σ , leaving the real-valued $(x - \mu)/\sigma$ as the interpretation of $(C_{\hat{x}}, e_{\hat{x}})$.

(5) Affine. The learnable (γ, e_γ) and (b, e_b) are integer parameters updated by the weight-update pipeline of Section 3.3; the final output is formed by one product and one sum: $y = \text{SUM}(\text{PROD}(\hat{x}, \gamma), b)$ using Equations (3), (5) and (6).

The backward pass caches $(C_\mu, C_{\sigma^2}, \text{inv}_\sigma, s_\sigma)$ and back-propagates through the five stages with only the product, sum, and shift-round primitives; no step needs floating point. In summary, the entire BN/LN layer uses integer sums, integer products, a single INT32 divide per reduction row, one integer square root, and shift-rounds.

Running statistics. The per-channel running mean and variance used at evaluation are stored as integer mantissa-exponent pairs $(\mu^{\text{run}}, e_{\mu^{\text{run}}})$, $(\sigma^{2,\text{run}}, e_{\sigma^{2,\text{run}}})$ updated by an exponential moving average

$$\mu^{\text{run}} \leftarrow (1 - \rho) \mu^{\text{run}} + \rho \mu. \quad (23)$$

The EMA factor ρ is represented as $\rho \approx \rho_{\text{num}}/2^{\rho_{\text{shift}}}$ (e.g. $\rho = 0.1 \approx 13/2^7$) and the update uses the same bounded-shift alignment policy as the velocity update, so the running statistics remain integer throughout training and inference. They are not updated through gradient descent, so they do not affect the training-state accounting in Table 2.

B.3 Output head: softmax and cross-entropy

From the final logits (X_L, e_{X_L}) of a batch of N samples over K classes, with $X_L \in \mathbb{Z}^{N \times K}$ and label vector $y \in \{0, \dots, K-1\}^N$, the classification head produces three quantities:

1. fixed-point softmax probabilities $p \in [0, 2^B - 2]^{N \times K}$ stored as unsigned intB ($B=8$, so $p \in [0, 254]$), used wherever a probability is consumed (e.g. the attention scores of Section B.4);
2. the scalar cross-entropy loss $\mathcal{L} = -\frac{1}{N} \sum_n \log(p_{n,y_n})$, used only for logging;

3. the loss gradient $\partial\mathcal{L}/\partial X_L = \frac{1}{N}(p - \mathbf{1}_y)$, which seeds the reverse-order layer loop of the training step.

All three share the two precomputed INT8 tables of Table 8; every other operation is an integer sum, product, divide-by-constant, or shift-round.

Table 8: Precomputed INT8 tables for the output head. Built once at initialization and shared across every output instance in the model.

LUT	Entry	Domain	Scale	Shape	Size
LUT _{exp}	$\text{ROUND}(\exp(k \cdot 2^{e_x}) \cdot 2^L)$	$k \in [-255, 0], \sim 30 \text{ exponents}$	$L=7$	$[30, 256]$	$\sim 8 \text{ KB}$
LUT _{log}	$\text{ROUND}(-\log(p/2^B) \cdot 2^{L_\ell})$	$p \in [1, 2^B - 2]$	$L_\ell=5$	$[256]$	$\sim 256 \text{ B}$

Softmax forward. Four integer steps per row:

1. $x'_i = x_i - \max_j x_j \in [-255, 0]$.
2. $E_i = \text{LUT}_{\text{exp}}[e_x, x'_i]$.
3. $Z = \sum_j E_j$ and $\text{inv}_Z = \lfloor 2^{S_R}/Z \rfloor$ with $S_R=30$ (one INT32 divide per row; the K normalizations below are multiplies).
4. $p_i = \text{SHIFT_ROUND}(E_i \cdot \text{inv}_Z, S_R - B)$, yielding unsigned $\text{int}B$ in $[0, 2^B - 2]$ (the 2^L scale of E_i and Z cancels in the ratio; stochastic rounding is applied during training).

The same cached $\text{int}B$ tensor p is reused by the backward and the loss below.

Loss value (reporting only). For logging only. For each sample we index $p_{n,y_n} \in [1, 2^B - 2]$ (floored to ≥ 1 so $-\log 0$ never appears) into LUT_{log} and take the integer batch mean

$$\mathcal{L} = \frac{1}{N} \sum_{n=1}^N \text{LUT}_{\text{log}}[p_{n,y_n}], \quad e_{\mathcal{L}} = -L_\ell, \quad (24)$$

with division by N via Equation (7) or Equation (8).

Gradient (seed of backpropagation). At the loss head, softmax and cross-entropy telescope into

$$(\partial\mathcal{L}/\partial X_L)_{n,k} = \frac{1}{N}(p_{n,k} - \mathbf{1}[k=y_n]), \quad (25)$$

a signed integer subtract on the $\text{int}B$ tensor p (promoted to $\text{int}(B+1)$ to carry the -1) followed by division by N .

Away from the loss (softmax inside attention), we apply the Jacobian on the cached p : $a = \sum_j p_j \partial y_j$ (product + sum), then $\partial x_i = p_i(\partial y_i - a)$ (subtract + product), with one compaction back to $\text{int}B$.

B.4 Vision Transformer block

The ViT reuses the same primitives as the CNN models: patch embedding is a strided convolution (integer product), and the CLS token and positional embeddings are learned integer parameters merged with the sum primitive. Each pre-norm block is

$$Y = X + \text{MHA}(\text{LAYERNORM}(X)), \quad Z = Y + \text{MLP}(\text{LAYERNORM}(Y)), \quad (26)$$

with LayerNorm from Section B.2, integer GELU inside the MLP from Section B.1, and the residual adds folded into the sum primitive (Equation (6)).

Multi-head attention, forward. Per head (hidden size d_h , sequence length T):

1. Three integer linear projections on the layer-normed input produce $(Q, e_Q), (K, e_K), (V, e_V)$ in $\mathbb{Z}^{T \times d_h}$.
2. Score accumulator $A_{QK} = QK^\top \in \mathbb{Z}^{T \times T}$ (INT32 product, Equation (5)), with natural exponent $e_Q + e_K$.

3. Scale by $1/\sqrt{d_h}$ and compact to $\text{int}B$:

$$(C_{\text{score}}, e_{\text{score}}) = \text{COMPACT}(\text{SHIFT_ROUND}(A_{QK} \cdot r_d, s_d), e_Q + e_K), \quad (27)$$

where (r_d, s_d) encode $1/\sqrt{d_h}$ as a rational shift (see next paragraph).

4. Row-wise integer softmax (Section B.3) yields $p \in [0, 2^{B_p}-1]^{T \times T}$.
5. Context accumulator $A_{\text{ctx}} = pV \in \mathbb{Z}^{T \times d_h}$, compacted to $\text{int}B$ with natural exponent $-B_p + e_V$.
6. A final integer linear concatenates heads and projects back to the embedding dimension.

Every intermediate is an (C, e) pair; only p and V need to be cached for the backward.

$1/\sqrt{d_h}$ as a rational shift. Since d_h is fixed at model-design time (commonly $d_h \in \{32, 64, 96\}$), $1/\sqrt{d_h}$ is a compile-time constant. We represent it as $r_d/2^{s_d}$ with $r_d = \lfloor 2^{s_d}/\sqrt{d_h} \rfloor$: for ViT-B ($d_h=64$), $s_d=10$ and $r_d=128$ give $128/1024 = 1/\sqrt{64}$ exactly; at the same s_d , $d_h=32$ and $d_h=96$ are lossless within one LSB. The backward applies the identical (r_d, s_d) to the score gradient.

Multi-head attention, backward. Given the upstream gradient ∂Z and the cached (p, V) , the reverse pass walks the forward pipeline in reverse, each stage using only product, sum, and shift-round primitives:

1. Output projection. Back through the final linear yields the context gradient $\partial A_{\text{ctx}} \in \mathbb{Z}^{T \times d_h}$ (one integer product) and the projection weight gradient.
2. Context matmul. The product $A_{\text{ctx}} = pV$ splits into $\partial p = \partial A_{\text{ctx}} V^\top$ and $\partial V = p^\top \partial A_{\text{ctx}}$.
3. Softmax Jacobian. Row-by-row on the cached p : $\partial C_{\text{score}, i} = p_i (\partial p_i - \sum_j p_j \partial p_j)$, exactly as in Section B.3.
4. Score scaling. The same rational shift is applied symmetrically: $\partial A_{QK} = \text{SHIFT_ROUND}(\partial C_{\text{score}} \cdot r_d, s_d)$.
5. Score matmul. $A_{QK} = QK^\top$ splits into $\partial Q = \partial A_{QK} K$ and $\partial K = \partial A_{QK}^\top Q$.
6. Q/K/V projections. Back through the three input projections produces input-gradient streams $\partial X^{(Q)}, \partial X^{(K)}, \partial X^{(V)}$ and weight gradients $\partial W_Q, \partial W_K, \partial W_V$.
7. Stream merge. Two applications of the sum primitive (Equation (6)) give $\partial X = \partial X^{(Q)} + \partial X^{(K)} + \partial X^{(V)}$.

Each INT32 accumulator is compacted back to $\text{int}B$ via Equation (3) before storage, so every tensor persisted across stages is an (C, e) pair.

C Training memory computation

The **Training memory (GB)** column of Table 3 reports the total training memory of each method at the benchmark configuration of its row. This appendix lists the per-tensor storage format of each method and the formula we use to combine it with the model’s parameter count and activation footprint.

Per-tensor storage format. The per-tensor formats assumed for each method are listed in Table 2 (main text). Summing the weight, gradient, and velocity widths gives the persistent state in bytes per parameter; the activation format scales with batch size and input resolution. Two points deserve comment. FP8 NVIDIA Micikevicius et al. [2022] quantizes to FP8 only at GEMM inputs; outputs and saved-for-backward activations remain FP16/BF16 (their §4.1 states “output tensors were left in higher precision. . . moving more tensors to FP8 is the subject of future study”). INT8 forward/backward methods Zhu et al. [2020], Zhao et al. [2021], Jin et al. [2022] and Ghaffari Ghaffari et al. [2022] are implemented as GPU emulations: the reduced-precision format is used at GEMM operand boundaries, while the tensor carried between layers (and saved for backward) is kept at FP32 in the implementation that produced their reported accuracy. The 8-bit claims in those papers refer to the arithmetic datapath, not to tensor storage.

Computing total training memory. For a model with P parameters and a forward pass that saves A activation elements per sample (summed across layers), training at batch size N uses

$$M_{\text{train}} = b_{\text{param}} \cdot P + b_{\text{act}} \cdot A \cdot N, \quad (28)$$

where b_{param} is the persistent state in bytes per parameter from Table 2 and b_{act} is the activation bytes per element from the same table. For the FP32 baseline, $b_{\text{param}} = 12$ and $b_{\text{act}} = 4$.

Per-configuration values. Table 9 lists P , A , and N for every benchmark configuration used in Table 3. The values of P are standard for the corresponding reference implementations; the activation counts A are obtained from PyTorch autograd profiling on the respective architectures, counting only the tensors that autograd retains for the backward pass.

Table 9: Per-configuration inputs to Equation (28): parameters P , saved-for-backward activations A per sample, batch size N .

Configuration	P (M)	A (M/sample)	N
ResNet-18 / CIFAR-10 (32^2)	11.2	1.29	256
MobileNetV2 / CIFAR-10 (32^2)	2.24	1.88	256
ViT-Ti / CIFAR-10 (32^2)	5.36	7.91	256
ViT-B / CIFAR-10 fine-tune (224^2)	86.0	35.1	64
ResNet-18 / CIFAR-100 (32^2)	11.2	1.29	256
ResNet-34 / CIFAR-100 (32^2)	21.3	2.01	256
ResNet-18 / ImageNet (224^2)	11.7	4.7	256
ResNet-50 / ImageNet (224^2)	25.6	21.5	256
ResNet-101 / ImageNet (224^2)	44.5	31.8	256

Plugging each (method, configuration) pair into Equation (28) with the bytes from Table 2 reproduces the **Training memory (GB)** entries of Table 3. The computation is implemented as a short Python script released with the paper.

D Training recipe and hyperparameters

This appendix collects the training details omitted from the main text: how inputs and activations are quantized, how biases and weight decay are folded into the integer pipeline, the integer learning-rate schedule, and the full hyperparameter grid of Table 10.

Hardware and software. All training runs use a single NVIDIA A100-SXM4 (40 GB) GPU. The integer pipeline is implemented on top of PyTorch 2.10 with CUDA 12.8 and cuDNN 9.1.

Input quantization. The dataloader produces float RGB tensors in $[0, 1]$; we normalize with standard per-channel statistics and then quantize once to INT8 at the framework boundary using the compaction primitive (Equation (3)):

$$(X_0, e_{X_0}) = \text{COMPACT}(\lfloor x \cdot 2^7 \rfloor, -7). \quad (29)$$

The shift is data-independent because the normalized range is bounded, so no online calibration is required. This is the only float-to-integer conversion in the pipeline; at evaluation the INT8 logits X_L feed integer arg max directly (Section B.3), so no integer-to-float conversion is needed either.

Activation quantization. Between layers, activations are re-quantized via the compaction primitive (Equation (3)); the shift S is computed from the per-tensor $\|A\|_\infty$ during the forward pass, so no separate calibration step is needed. For ViT-B we additionally clip activations at the 99.99th percentile of absolute values (`scale_percentile=0.9999`) before computing the shift, preventing a single rare outlier from consuming the dynamic range. CNNs and ViT-Ti do not require this clipping at any of the bit widths we tested.

Biases (ViT only). The CNN models in our setup (ResNet-18, MobileNetV2) omit convolution and linear biases because batch normalization absorbs them; ViT-B retains biases in its Q/K/V and MLP projections. Each bias is stored as a (b, e_b) pair at the weight bit width (INT8 by default) and aligned to the output exponent e_{XW} via the bounded-shift policy of Section 3.2.2 before being added to the pre-activation.

Momentum and weight decay. The momentum coefficient β is stored as the integer pair $(\beta_{\text{num}}, \beta_{\text{shift}})$ with $\beta \approx \beta_{\text{num}} 2^{-\beta_{\text{shift}}}$; the default $\beta=0.9$ uses $(\beta_{\text{num}}, \beta_{\text{shift}})=(230, 8)$ giving $\beta \approx 230/2^8$. L_2 regularization is injected into the per-group gradient *before* the momentum update, so the decay term participates in the velocity accumulation:

$$G_k \leftarrow G_k + \text{SHIFT_ROUND}(\lambda_{\text{num}} \cdot W_k, \lambda_{\text{shift}}), \quad (30)$$

with $(\lambda_{\text{num}}, \lambda_{\text{shift}})$ precomputed for the target λ (e.g. $\lambda=10^{-4}$). The augmented G_k then feeds Equation (13), so no separate weight-decay step is applied at the weight update itself.

Integer learning-rate schedule. The learning rate is stored as $\eta \approx \eta_{\text{num}} 2^{-\eta_{\text{shift}}}$. Since schedules are known before training, we precompute one $(\eta_{\text{num}}^{(t)}, \eta_{\text{shift}}^{(t)})$ pair per iteration t as the best-fit

$$(\eta_{\text{num}}^{(t)}, \eta_{\text{shift}}^{(t)}) = \arg \min_{\eta_{\text{num}}, \eta_{\text{shift}}} |\eta_{\text{num}} 2^{-\eta_{\text{shift}}} - \eta^{(t)}| \quad \text{subject to} \quad 1 \leq \eta_{\text{num}} \leq \eta_{\text{num}, \text{max}}, \quad (31)$$

where $\eta^{(t)}$ follows the regime in use: geometric decay $\gamma\eta$ at milestones for step schedules; $\eta_{\text{min}} + \frac{1}{2}(\eta_{\text{max}} - \eta_{\text{min}})(1 + \cos(\pi t/T))$ for cosine; linear interpolation during warmup.

Table 10: Per-model hyperparameters used for Table 3. All runs: SGD with momentum, cosine LR with linear warmup.

	RN-18 C10/C100	MNv2 C10	ViT-Ti C10	ViT-B C10	RN-18 INet
Batch size	256	256	256	64	256
Epochs	200	200	400	50	90
Base LR	0.1	0.05	0.05	0.001	0.1
Warmup epochs	5	5	5	0	5
Momentum β	0.9	0.9	0.9	0.9	0.9
Weight decay	10^{-4}	10^{-4}	10^{-4}	10^{-4}	10^{-4}
Velocity bits B_V	8	8	8	8	8
Accumulator bits	32	32	32	32	32
Activation clip pct.	—	—	—	0.9999	—
Augmentation	crop+flip	crop+flip +cutout	crop+flip +AutoAug	RandAugment	crop+flip +AutoAug

E Additional experiments

This appendix supplements Section 4 with: (i) training dynamics for ResNet-18/CIFAR-10 (Figure 3); (ii) an operator-level breakdown of the ViT-B accuracy gap on CIFAR-10 (Table 11); and (iii) a layer-by-layer cosine-similarity analysis of the INT8 ViT that motivates the need for fine-tuning.

Training dynamics. Figure 3 shows that INT8 training tracks the FP32 baseline on ResNet-18/CIFAR-10 over 200 epochs; step-LR drops produce clean discrete transitions with no instability or divergence.

Where the INT8 ViT drifts. Table 11 attributes the entire 3.2% pre-finetuning gap to the Linear projections: replacing only the Linear layers with INT8 (keeping LayerNorm, GELU, and softmax in float) costs 3.2%, while subsequently moving LayerNorm, GELU, and softmax into the integer pipeline costs nothing further (LayerNorm -0.1 , GELU $+0.1$, softmax $+0.2$). In other words, the integer attention machinery introduced in this paper – LUT-based softmax, $1/\sqrt{d_h}$ as a rational shift, and the cached- p Jacobian backward – is essentially free in accuracy terms; the residual gap is the generic Linear-quantization error that any INT8 fwd/bwd method incurs.

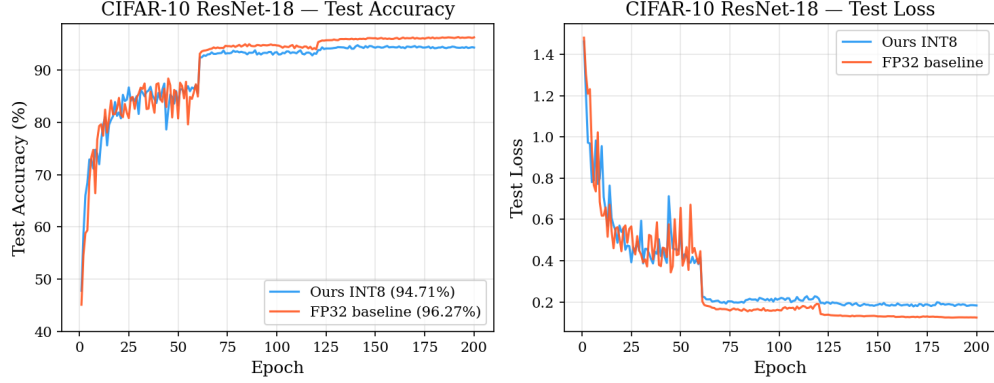


Figure 3: ResNet-18 / CIFAR-10: integer-only (INT8) vs. FP32 baseline, 200 epochs, step LR at 60/120/180.

Table 11: Operator layer ablation on ViT-B fine-tuning (CIFAR-10). All rows use INT8 weights and activations.

Softmax	GELU	LayerNorm	Linear	Acc. (%)	Δ vs. float
float	float	float	float	99.1	(float baseline)
float	float	float	INT8	95.9	−3.2
float	float	INT8	INT8	95.8	−3.3
float	INT8	INT8	INT8	95.9	−3.2
INT8	INT8	INT8	INT8	96.1	−3.0
Ghaffari et al. [2022] (float softmax, FP32 weights)				98.8	−0.3

A complementary cosine-similarity analysis of the INT8 ViT without fine-tuning shows where this Linear-quantization error compounds. The match to the float model degrades through the stack from 0.977 after the patch embedding to 0.83 after block 0, 0.64 after block 2, and below 0.01 after block 5. The drop is concentrated inside the attention sublayer – 0.95 $\rightarrow$ 0.64 in a single block – but, consistent with Table 11, the source is not the attention-specific operators: it is the four Linear projections per block (QW_Q , KW_K , VW_V , and the output projection), whose ~ 72 INT8 compaction points compound multiplicatively across the 12-block stack. Fine-tuning then closes the gap as the weights adapt to the INT8 quantizer.