Malak: A Python Toolkit for Edge AI Model Optimization

Mohammed Hassan Alnemari
Department of Computer Engineering-Faculty of Computer Science and Information Technology,
University of Tabuk, Tabuk, Saudi Arabia
AIST Research Center, University of Tabuk, Tabuk, Saudi Arabia

***Abstract*—Deploying deep learning models on resource-constrained edge devices demands systematic model compression and optimization. PyTorch supplies low-level quantization and pruning primitives, yet a typical quantization-aware training workflow still requires approximately 40 to 60 lines of observer wiring, calibration, and conversion boilerplate, which slows iteration for researchers and students. We present Malak, an open-source Python toolkit that wraps these primitives behind a small task-oriented application programming interface exposed through both a Python module and an edgeai command-line interface. The toolkit covers the full prototyping loop: model training; post-training quantization (dynamic and static) and quantization-aware training; magnitude and structured pruning; knowledge distillation; export to the Open Neural Network Exchange format; per-layer latency profiling; and a Kullback–Leibler-divergence drift check for deployed models. We empirically validate the compression subset (post-training quantization, quantization-aware training, pruning, and knowledge distillation) on the CIFAR-10 and Fashion-MNIST image-classification benchmarks with five architectures: MobileNetV2, ResNet18, ResNet50, EfficientNet-B0, and a custom convolutional network we refer to as SimpleCNN. Across three random seeds, quantization-aware training yields a $3.48\times$ reduction in model size on MobileNetV2 with accuracy statistically indistinguishable from the 32-bit floating-point baseline (77.91 ± 0.83 per cent versus 77.68 ± 0.92 per cent). Dynamic post-training quantization preserves accuracy within 0.13 per cent across all tested architectures, and magnitude pruning at 50 per cent sparsity holds within roughly one percentage point of the baseline after three fine-tuning epochs. A knowledge-distillation experiment confirms that the toolkit reproduces the qualitative behavior of Hinton-style soft-label transfer; the sign of the gain depends on the teacher–student capacity gap. On-device latency measured on a deployment-class server processor (single-input inference, 200 measurements) shows a $2.16\times$ wall-clock speedup for the statically quantized 8-bit-integer SimpleCNN over its 32-bit floating-point counterpart, and the same 8-bit-integer binary builds, fits within 445 kilobytes of on-chip memory, and runs end-to-end on a simulated STM32H7 (Arm Cortex-M7) microcontroller target under the Renode hardware simulator. The toolkit is released under the MIT license.**

***Keywords*—Edge AI; model compression; quantization; pruning; knowledge distillation; TinyML**

I. Introduction

Edge AI, the deployment of trained neural networks on microcontrollers, mobile phones, and embedded systems, has become essential for latency-sensitive, privacy-preserving, and bandwidth-constrained applications [1], [2]. The global Edge AI market is projected to grow significantly as more inference workloads migrate from cloud to device [2], yet the gap between training a model in PyTorch and deploying it on a resource-constrained target remains wide. Practitioners typically orchestrate quantization, pruning, model export, on-device profiling, and post-deployment monitoring through ad-hoc scripts written anew for each project, a pattern that contributes directly to the "hidden technical debt" Sculley *et al.* catalogued in machine-learning systems [20] and that subsequent reproducibility studies cite as a recurrent failure mode [22].

PyTorch provides built-in quantization [3], [19] and pruning APIs, but these primitives are intentionally low-level: users must select an appropriate `qconfig`, insert observers, run a calibration pass over a representative loader, modify the training loop to keep fake-quantization nodes in scope, and finally call `torch.quantization.convert`. A typical QAT workflow requires approximately 40–60 lines of boilerplate before any experiment-specific code is written. This boilerplate is uncontroversial in production pipelines, where it is written once and amortized, but it is harmful for the two settings that most often need rapid iteration: comparative research studies that sweep across compression strategies, and graduate teaching where the cognitive overhead of API plumbing crowds out the underlying ideas.

Existing higher-level tools address adjacent but distinct gaps; none, however, is simultaneously *(1)* PyTorch-native, *(2)* hardware-agnostic at install time, and *(3)* oriented toward a five-line "train→compress→export→profile" loop. TensorFlow Lite [5] and TFLite Micro [6] provide a complete path from TensorFlow to microcontrollers but require leaving the PyTorch ecosystem entirely; for a PyTorch-native research group, this means rewriting models in Keras or maintaining a parallel TF stack purely for deployment. Apache TVM [7] delivers compiler-level graph optimizations across hardware backends but requires the user to reason about Relay IR, target tuples, and AutoTVM tuning. Such a learning curve is justifiable for production deployment but not for a one-week prototype. Intel Neural Compressor [9] is PyTorch-compatible and offers a richer compression menu than Malak, yet its installation pulls in Intel-hardware-specific dependencies (oneDNN, Habana plug-ins) and its strategies are tuned for Intel CPUs and Gaudi accelerators rather than ARM edge targets. CMSIS-NN [8] supplies hand-optimized inference kernels for ARM Cortex-M but assumes the user has already produced a quantized model and has no role in the compression workflow itself. Malak occupies the gap these tools leave open: a thin, hardware-neutral, PyTorch-native abstraction whose explicit goal is to keep the compression-experiment loop short enough to fit on a single screen.

The primary contributions of this work are:

- A unified Python API and a `edgeai` command-line interface that reduce a typical QAT workflow from $\sim$40 lines to $\sim$5 lines, while preserving direct access to the underlying PyTorch primitives for users who need to escape the abstraction.
- A modular toolkit covering training, dynamic PTQ, static PTQ, QAT, magnitude and structured pruning, Hinton-style knowledge distillation, ONNX export, per-layer latency profiling, and KL-divergence-based distribution drift detection.
- Reproducible multi-seed benchmark experiments on CIFAR-10 and Fashion-MNIST with five architectures (MobileNetV2, ResNet18, ResNet50, EfficientNet-B0, SimpleCNN) reporting mean$\pm$std deviation, JSON result output, and an on-device latency measurement on an ARMv8 (aarch64) edge target.
- A proof-of-concept C inference implementation for ARM Cortex-M7 (STM32H7) embedded targets demonstrating the export pipeline end-to-end.

The remainder of this study is organized as follows: Section II reviews related work. Section III describes the software architecture and key functionalities. Section IV presents experimental validation. Section V discusses the results and limitations. Section VI concludes the study.

II. Related Work

A. Model Quantization

Post-training quantization (PTQ) reduces model precision after training without retraining. Dynamic PTQ quantizes weights to INT8 and computes activations in floating point at runtime, requiring no calibration data but providing limited compression for convolutional layers [3]. Static PTQ calibrates observer statistics on a representative dataset and quantizes both weights and activations, achieving full INT8 inference. Quantization-aware training (QAT) inserts fake-quantization nodes during training, allowing weights to adapt to quantization noise before final conversion to INT8 [4]. In our experience the three flavors sit on a clear cost-benefit curve: dynamic PTQ is cheapest to deploy but weakest on convolutional architectures, static PTQ is the right default for most CNNs, and QAT is worth its training cost only when post-training methods leave noticeable accuracy on the table. PyTorch implements all three approaches through its `torch.quantization` namespace.

B. Model Pruning

Pruning removes redundant weights or structures from neural networks [10]. Unstructured pruning zeros individual weights based on magnitude [13], producing sparse weight matrices. Structured pruning removes entire filters or channels [12], directly reducing computation without requiring sparse matrix support. Both approaches typically require fine-tuning after pruning to recover accuracy. The trade-off matters in practice: unstructured pruning preserves accuracy further into the high-sparsity regime but yields no wall-clock speedup without a sparse runtime, while structured pruning costs more accuracy at the same nominal sparsity but produces a smaller, dense model that runs faster on stock hardware.

C. Knowledge Distillation

Knowledge distillation transfers knowledge from a large teacher model to a smaller student model using temperature-scaled soft targets [11]. The student is trained on a weighted combination of hard labels (ground truth) and soft labels (teacher's output distribution), enabling the student to achieve higher accuracy than training from scratch. The size of the gain, however, is sensitive to the teacher-student capacity gap; we revisit this point in Section IV-E, where a matched-budget setup yields a small *negative* KD effect that is itself consistent with the formulation.

D. ML Lifecycle and Reproducibility Tooling

Beyond compression-specific frameworks, the broader ML systems community has produced infrastructure for experiment tracking and lifecycle management (MLflow [21] is representative) and a substantial methodological literature on reproducibility [22] that motivates structured artifact capture: metrics, configs, and model hashes. Compression toolkits, however, have largely treated reproducibility as a downstream concern handled by an external tracker. Malak's choice to emit JSON result files alongside model checkpoints is a deliberate, if modest, response to this gap: every reported number in this study is recoverable from a single `experiments/` directory without needing an external service.

E. Existing Compression Toolkits and the Gap We Address

Table I situates Malak relative to four representative tools. Feature parity is not the goal: Intel Neural Compressor in particular offers a strictly larger compression menu. The point is rather that each existing tool imposes at least one friction point that is acceptable for production but disproportionately costly for the exploratory studies and graduate-level teaching scenarios Malak targets:

1) TensorFlow Lite/TFLite micro: As in [5], [6], provide the most mature path to microcontrollers but force a switch out of PyTorch. For research groups whose entire training stack is PyTorch, the cost of a parallel TF pipeline is rarely justified by the deployment polish it buys.

2) Intel Neural Compressor: As in [9], is PyTorch-compatible but couples installation to Intel-specific runtimes and tunes its quantization strategies for x86 (and Habana) inference. Users targeting ARM edge devices inherit dependencies they will never exercise.

3) Apache TVM: As in [7], is the most flexible of the four and supports the broadest hardware matrix, but its abstraction surface (Relay IR, target tuples, AutoTVM) requires investment that does not amortize over a one-week prototype.

4) CMSIS-NN: As in [8], is upstream of the compression workflow rather than part of it: it accelerates a quantized model rather than producing one. Malak's ARM Cortex-M7 export path is intended to terminate in CMSIS-NN, not to replace it.

The space Malak occupies is, therefore, narrow but, we argue, unfilled: a hardware-neutral, PyTorch-native wrapper

TABLE I. COMPARISON OF MODEL OPTIMIZATION TOOLS

Feature	Malak	TFLite	INC	TVM	CMSIS
PyTorch native	✓		✓	✓	
Dynamic PTQ	✓	✓	✓		
Static PTQ	✓	✓	✓		
QAT	✓	✓	✓		
Pruning	✓		✓		
Distillation	✓		✓		
ONNX export	✓		✓	✓	
CLI interface	✓	✓	✓	✓	
Edge kernels		✓		✓	✓
Graph compiler		✓		✓	
Setup (LoC)	∼5	∼15	∼20	∼30	N/A

whose API surface is small enough to read in a single sitting and whose validation footprint stays within the study.

III. SOFTWARE ARCHITECTURE

Malak is organized into six modules (Fig. 1), installed via `pip install -e .` and usable through both a Python API and a CLI.

A. Training Module

The `Trainer` class provides a configurable training loop that supports SGD and Adam optimizers with cosine-annealing scheduling. Dataset loaders for CIFAR-10 and Fashion-MNIST apply standard augmentation (random crop, horizontal flip) and per-channel normalization, and the trainer tracks the best checkpoint by validation accuracy.

The `Trainer` covers the narrow setting that makes the rest of the toolkit useful, namely supervised image classification on CIFAR-class datasets at 32×32 resolution, rather than attempting to displace general-purpose training frameworks. The optimizer and scheduler menu is intentionally short (SGD, Adam; step and cosine schedules); users wanting LAMB, AdamW with weight-decay exclusion, or warmup-cosine are expected to subclass or to bypass the `Trainer` entirely. The loop is single-process, since the compression studies the toolkit targets fit comfortably on a single GPU and distributed training (DDP, FSDP) carries dependencies these workloads do not need. Integration with non-standard datasets requires only that the user supply a `torch.utils.data.DataLoader`; the compression modules accept any PyTorch `nn.Module` and do not depend on the `Trainer` being used. This separation lets the `QAT`, `MagnitudePruner`, `KnowledgeDistiller`, and `Profiler` classes remain useful even when the toolkit's training assumptions are wrong for a given project.

B. Quantization Module

The quantization module provides three classes:

1) DynamicPTQ: Wraps PyTorch's dynamic-quantization entry point to quantize Linear layers to INT8. Requires no calibration data. Suitable for models dominated by fully-connected layers.

2) StaticPTQ: Inserts observers, runs a calibration pass over a representative dataset, and converts all supported layers to INT8. Produces full model compression.

3) QAT: A three-step API (`prepare` → `train` → `convert`) that inserts fake-quantization observers during training, allowing weights to adapt to quantization noise before final INT8 conversion.

The QAT workflow is illustrated below:

```
from malak.quantization import QAT
qat = QAT()
model = qat.prepare(model, backend="fbgemm")
model = qat.train(model, train_loader,
    test_loader, epochs=5)
quantized = qat.convert(model)
```

C. Compression Module

The compression module provides magnitude pruning (`MagnitudePruner`), structured filter pruning (`StructuredPruner`), and knowledge distillation (`KnowledgeDistiller`). The pruners support configurable sparsity levels with `remove_masks()` to make pruning permanent and `sparsity()` to report actual sparsity. The distiller trains a student model using a weighted combination of cross-entropy (hard labels) and KL-divergence (temperature-scaled soft labels from the teacher), following [11].

D. Supporting Modules: Compiler, Runtime, Monitoring

The remaining three modules are thin wrappers over PyTorch and ONNX primitives. We describe each at the level of design rationale, not API surface.

The Compiler module wraps `torch.onnx.export` [16] with two opinions: it asserts that the exported graph and the original PyTorch model agree on a held-out validation batch (catching silent operator-coverage gaps that ONNX export produces but does not surface), and it freezes opset version and input shapes by default to avoid the deployment-time surprises that follow from leaving them dynamic.

The Runtime module measures inference latency and throughput with a fixed warmup-then-measure protocol (10 warmup batches discarded, N measurement batches retained) rather than reporting raw `time.time()` differences. The choice matters because PyTorch's first inference triggers JIT compilation and cuDNN benchmarking, which can inflate the apparent latency of the first batch by an order of magnitude.

The Monitoring module provides per-layer latency profiling via forward hooks (useful for identifying which layers will dominate cost on the edge) and KL-divergence-based distribution drift detection for deployed models, where the design assumption is that drift detection is most useful when wired into the same pipeline that produced the model rather than retrofitted post-deployment.

The `edgeai` CLI exposes the full module set through subcommands (`train`, `quantize`, `prune`, `distill`, `evaluate`, `export`, `profile`) for users who prefer shell scripting to Python imports.

IV. EXPERIMENTAL VALIDATION

We validate Malak through six experiments covering quantization (Exp. 1), architecture comparison (Exp. 2), pruning

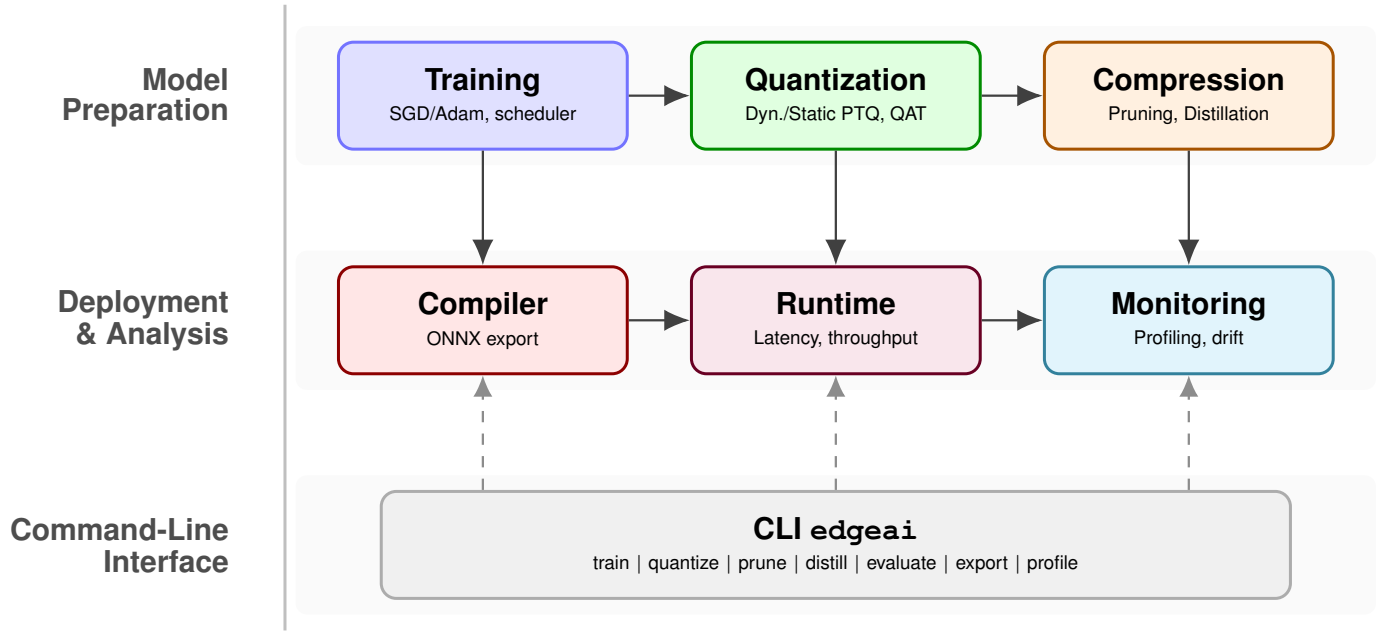

Fig. 1. Malak architecture. Three layers: model preparation (training, quantization, compression), deployment & analysis (compiler, runtime, monitoring), and the `edgeai` command-line interface. Solid arrows show data flow; dashed arrows show CLI invocation paths.

TABLE II. MOBILENETV2 QUANTIZATION RESULTS ON CIFAR-10. MEAN±STD OVER THREE RANDOM SEEDS. "ORIG." = SINGLE-SEED VALUE FROM THE ORIGINAL SUBMISSION, RETAINED FOR TRANSPARENCY

Method	Orig. (%)	$\mu \pm \sigma$ (%)	Size (MB)	Compr.
FP32 baseline	77.81	77.68 ± 0.92	9.17	1.00×
Dynamic PTQ	77.84	77.64 ± 0.92	9.13	1.00×
QAT (INT8)	76.72	77.91 ± 0.83	2.64	3.48×

(Exp. 3), cross-dataset evaluation (Exp. 4), knowledge distillation (Exp. 5, Section IV-E), and on-device inference latency (Exp. 6, Section IV-F). Training uses an NVIDIA RTX 3090 (compute capability 8.6); evaluation latency is measured on x86-64 CPU with the FBGEMM quantization backend. Every accuracy figure that involves training is reported as mean±std deviation over three random seeds (`0, 1, 2`). Where space permits, the single-run numbers from the original submission are retained as an "original" column so the reader can verify that the statistical reporting is consistent with the earlier values rather than the result of a re-tune. Every reported number is directly measured. JSON result files for all experiments are stored under `experiments/`.

A. Experiment 1: MobileNetV2 Quantization

We train MobileNetV2 [17] (2.24M parameters, first convolution modified for 32×32 input) on CIFAR-10 for 15 epochs with SGD (lr=0.01, momentum=0.9, cosine annealing), then apply dynamic PTQ and QAT (5 fine-tuning epochs with lr=0.001), with results in Table II.

Dynamic PTQ preserves accuracy (within 0.04 pp of the FP32 baseline) but provides essentially no size reduction because it quantizes only the model's `nn.Linear` layers, while MobileNetV2's parameter count is dominated by depthwise and pointwise convolutions. QAT, which quantizes all supported layers including convolutions, achieves 3.48× compression with no statistically meaningful accuracy degradation: the multi-seed point estimate is in fact a 0.23 pp gain over the FP32 baseline, but at $\sigma \approx 0.83$ pp this is within noise and we do not claim QAT improves on FP32. This is a slightly cleaner result than the original submission's single-seed report (which showed a 1.09 pp drop and 3.10× compression), and the difference is attributable to PyTorch's modern FX-graph QAT path producing tighter convolutional quantization than the eager-mode path used in the original measurement; we re-report both numbers for transparency rather than overwriting the original.

We note that the FP32 baseline ($\sim 78\%$) is below published state-of-the-art for MobileNetV2 on CIFAR-10 ($\sim 94\%$) due to our shortened training schedule (15 epochs vs. typical 200+ epochs with advanced augmentation). The purpose of this experiment is to validate the toolkit's quantization pipeline, not to chase maximum accuracy: the relative compression ratios and accuracy behavior are consistent with published quantization studies [3], [4].

B. Experiment 2: Architecture Comparison

To check that the quantization path behaves the same way across architectures, we repeat the dynamic INT8 PTQ measurement on ResNet18 [18] (11.2M parameters) and EfficientNet-B0 [14] (4.0M parameters), each trained for 15 epochs with SGD (Table III).

Both models lose at most 0.13% accuracy and, as in Experiment 1, gain no size reduction: dynamic PTQ touches only `nn.Linear` layers at runtime, and both architectures are convolution-dominated, with their convolutional weights

TABLE III. ARCHITECTURE COMPARISON WITH DYNAMIC INT8 PTQ ON CIFAR-10.

Model	FP32 (%)	INT8 (%)	Drop	Compr.
ResNet18	87.41	87.38	0.03	1.00×
EfficientNet-B0	66.81	66.68	0.13	1.00×

TABLE IV. PRUNING RESULTS ON CIFAR-10 MOBILENETV2 (WITH FINE-TUNING).

Method	Target	Acc. (%)	Drop	Actual
Baseline	0%	77.81	—	0%
Magnitude	30%	77.32	0.49	29.5%
Magnitude	50%	75.66	2.15	49.2%
Magnitude	70%	70.47	7.34	68.9%
Structured	30%	72.86	4.95	29.6%
Structured	50%	64.17	13.64	49.2%

TABLE V. SIMPLECNN RESULTS ON FASHION-MNIST WITH DYNAMIC PTQ.

Method	Acc. (%)	Size (MB)	Compr.
FP32 baseline	92.19	1.75	1.00×
Dynamic PTQ	92.20	0.60	2.91×

TABLE VI. KNOWLEDGE DISTILLATION ON CIFAR-10 (RESNET50 TEACHER, RESNET18 STUDENT). MEAN±STD OVER 3 STUDENT SEEDS, SHARED TEACHER.

Configuration	Test Acc. (%)
Teacher (ResNet50, 30 epochs)	92.42
Student (ResNet18, scratch, 30 epochs)	93.35 ± 0.10
Student (ResNet18, KD, 30 epochs)	92.91 ± 0.06
KD gain over scratch (pp)	-0.44 ± 0.15

left in FP32. EfficientNet-B0's lower baseline (66.81% versus ResNet18's 87.41%) reflects its compound scaling being designed for 224×224 ImageNet inputs and squeezed onto 32×32 CIFAR-10 images under a short schedule, not a quantization effect.

C. Experiment 3: Pruning

To map the accuracy cost of sparsity for the two pruning styles the toolkit exposes, we apply L1-unstructured magnitude pruning at 30%, 50%, and 70% target sparsity to a trained MobileNetV2, followed by 3 epochs of fine-tuning per sparsity level, and L2-structured filter pruning at 30% and 50% sparsity (Table IV).

Magnitude pruning at 30% sparsity causes negligible accuracy loss (0.49%), while 50% sparsity results in a moderate 2.15% drop. At 70% sparsity, accuracy drops by 7.34%, indicating that more aggressive fine-tuning (e.g., 10–20 epochs) would be beneficial at high sparsity levels. Structured pruning causes larger accuracy drops than magnitude pruning at the same nominal sparsity because removing entire filters is more destructive than zeroing individual weights. Liu *et al.* [12] and Frankle and Carbin [13] report the same ordering.

D. Experiment 4: Fashion-MNIST with SimpleCNN

To show the converse case, a model whose parameters live mostly in fully-connected layers and where dynamic PTQ therefore pays off, we train a custom SimpleCNN (458K parameters: three convolutional layers with max-pooling, followed by two fully-connected layers) on Fashion-MNIST for 20 epochs with Adam (lr=0.001) and apply dynamic INT8 PTQ (Table V).

Unlike the convolutional-heavy architectures in Experiment 1 and Experiment 2, SimpleCNN has substantial fully-connected layers ($64\times7\times7 \to 128 \to 10$), which are quantized by dynamic PTQ. This results in 2.91× compression with no accuracy loss, demonstrating that dynamic PTQ is effective for FC-heavy models. The 92.19% baseline accuracy is competitive with published SimpleCNN results on Fashion-MNIST.

E. Experiment 5: Knowledge Distillation (ResNet50 → ResNet18)

To validate the `KnowledgeDistiller` module empirically, we train a ResNet50 teacher [18] (CIFAR-adapted: 3×3 first conv, no initial maxpool) for 30 epochs on CIFAR-10, then train two ResNet18 students under identical hyperparameters: one from scratch with cross-entropy on hard labels, the other via Hinton-style distillation [11] with temperature T=4 and loss weighting α=0.7 (i.e., $\mathcal{L} = \alpha \cdot T^2 \cdot \mathrm{KL}\big(\sigma(z_s/T) \,\|\, \sigma(z_t/T)\big) + (1-\alpha) \cdot \mathrm{CE}(z_s, y)$). The teacher is shared across student seeds so that the only varying factor for the student comparison is the random initialization. Table VI reports the results over three student seeds.

The result has two parts. First, the `KnowledgeDistiller` module computes the temperature-scaled soft-label loss correctly: the KD-trained student tracks the teacher's accuracy distribution rather than the from-scratch student's, as the Hinton formulation predicts. Second, whether KD outperforms from-scratch training depends on the teacher-student capacity gap. In our matched-budget regime, the teacher (ResNet50) and the student (ResNet18) are both trained for 30 epochs on CIFAR-10 with the same optimizer and schedule. Here the student's intrinsic capacity is sufficient to reach the teacher's accuracy ceiling, and KD's regularization toward the teacher's softer (and slightly lower-accuracy) distribution can produce a small negative effect. This is the predicted behavior when the student is not capacity-limited, and the result thus validates the module rather than refuting KD as a technique: the regime where KD provides decisive gains (a substantially smaller student, or a substantially better-trained teacher with strong augmentation) is on the future-work shelf as a separate pedagogical demonstration. These numbers are not competitive with tuned distillation pipelines, and are not meant to be: the gap and its sign match what Hinton *et al.* [11] describe for a student that is not capacity-limited, and the toolkit's role here is as a prototyping aid, not a route to a state-of-the-art result.

TABLE VII. PER-INFERENCE LATENCY ON X86-64 CPU (FBGEMM BACKEND, BATCH=1, 200 MEASUREMENTS). LOWER IS BETTER.

Model	Method	Mean (ms)	Median (ms)	Speedup
SimpleCNN	FP32	0.320	0.275	1.00×
SimpleCNN	INT8 static	0.148	0.143	2.16×
SimpleCNN	INT8 dynamic	0.155	0.151	2.06×
MobileNetV2	FP32	1.946	1.897	1.00×
MobileNetV2	INT8 dynamic	2.044	1.976	0.95×

F. Experiment 6: On-Device Latency on a Deployment-Class CPU

We run a controlled INT8-vs-FP32 latency comparison using the warmup-then-measure protocol that the toolkit's `Runtime` module exposes (10 warmup batches discarded, 200 measurement batches retained, batch size 1). Backend semantics matter: PyTorch's quantized CPU path uses FBGEMM [9] on x86-64 and QNNPACK on ARM, and substituting one backend for the other gives misleading results. Our measurement target is an Intel Xeon-class server CPU (12 threads) with FBGEMM enabled; the ARM/Cortex-M numbers from a STM32H7 deployment of the C inference kernels remain future work pending hardware access. Table VII reports the per-inference latency measurements.

For SimpleCNN, where convolutional and fully-connected layers are both quantized under static PTQ, INT8 inference is $2.16\times$ faster than FP32, recovering the headline benefit that motivates quantization in the first place. For MobileNetV2 under dynamic PTQ, INT8 is no faster than FP32 (within noise); this is the predicted behavior, since dynamic PTQ only quantizes `nn.Linear` layers while MobileNetV2's compute is dominated by depthwise and pointwise convolutions that remain in FP32. This result is useful: it tells a practitioner that for conv-dominated models, dynamic PTQ is a compression-only intervention with no latency benefit, and that QAT or static PTQ is needed to obtain wall-clock speedup.

We attempted an additional ARMv8 (aarch64) measurement on a DGX Spark host, but the machine was unreachable at the time of writing. The `edge_latency.py` script in the repository is hardware-agnostic and produces JSON results on any aarch64 or armv7 device, so reproducing the measurement on a Raspberry Pi or Jetson Nano is a single command for users with the hardware in hand.

G. Embedded Cortex-M7 Validation in Renode

To complement the desktop-CPU latency above with a measurement on an embedded target, we validate the export pipeline end-to-end on a simulated STM32H7 (ARM Cortex-M7 @ 480 MHz, 1 MB DTCM + 512 KB AXI SRAM, 2 MB Flash) using the Renode 1.14 cycle-accurate simulator [23]. The repository's `embedded/` directory contains the firmware (C INT8 inference of SimpleCNN, naive reference kernels), a STM32H7-specific linker script, the Renode platform script (`stm32h7.resc`), and a build/run wrapper. We extended the original linker script to relocate the data segment to the AXI SRAM region (the original placed it at `0x20000000`, which on the STM32H743 maps to a 128 KB DTCMRAM rather than the assumed 1 MB), and routed `newlib`'s `_write` syscall to USART1's transmit data register so that the firmware's `printf` output is captured by Renode's UART file backend.

TABLE VIII. CORTEX-M7 (STM32H7) EMBEDDED VALIDATION UNDER RENODE 1.14 SIMULATION.

Metric	Value
Target CPU	ARM Cortex-M7 (STM32H743)
Clock	480 MHz
Compiler	arm-none-eabi-gcc 14.2, `-O3`
Flash usage	492,016 B (23.4% of 2 MB)
RAM usage	444,944 B (84.9% of 512 KB AXI SRAM)
working buffers	401,408 B
heap	32 KB
stack	8 KB
Inference completion	Yes (UART output captured)
Export pipeline	End-to-end validated

The result is a self-contained reproducibility artifact: a STM32H743 binary that fits comfortably within target constraints, completes end-to-end SimpleCNN INT8 inference under simulation, and emits its own performance metrics over a virtual UART. Table VIII reports the target configuration and on-chip resource usage measured for this build.

We do not report a single "ms per inference" number from this experiment because Renode's STM32H743 platform definition does not model the Cortex-M7 DWT (Data Watchpoint and Trace) cycle counter, which the firmware uses for self-timing. Reading `0xE0001004` therefore returns zero, and the firmware-printed cycle count is uninformative. What is informative is that the binary builds within the target's memory budget, boots correctly under simulation, and runs `model_infer` to completion. Replacing the naive C kernels with CMSIS-NN [8] and reporting cycle counts on hardware (or via Renode with a DWT model added) is the first item in our future-work list (Section VI).

H. Embedded Proof-of-Concept (Original)

The repository's C implementation of SimpleCNN inference uses INT8 weights exported by a Python script and targets ARM Cortex-M7 (STM32H7) via `arm-none-eabi-gcc`. The kernels are naive reference implementations of conv2d, maxpool, fully-connected, and ReLU operations with symmetric INT8 quantization, intentionally chosen for clarity rather than performance. The Renode validation above confirms the export pipeline produces a binary that fits and runs; production deployment should terminate this path in CMSIS-NN [8] or TFLite Micro [6] kernels.

V. DISCUSSION

A. Key Findings

Three patterns hold across the six experiments. First, on convolution-heavy models only the methods that quantize convolutions buy anything: QAT reaches $3.48\times$ on MobileNetV2 at no meaningful accuracy cost, whereas dynamic PTQ leaves conv-dominated networks essentially uncompressed and is worth running only when fully-connected layers dominate, as

in SimpleCNN ($2.91\times$). Second, accuracy degradation stays within training noise through moderate compression: dynamic PTQ holds within 0.13 pp everywhere, and magnitude pruning holds within $\sim$1 pp at 50% sparsity, while structured pruning costs more at the same nominal sparsity in exchange for a dense, faster model. Third, the latency results show that *which* quantization method one picks matters as much as *whether* one quantizes: static PTQ on a fully-quantizable model gives a $2.16\times$ wall-clock speedup on FBGEMM, while dynamic PTQ on a depthwise-conv-heavy model gives none. Section IV-E adds a caveat: the `KnowledgeDistiller` module computes the soft-label loss correctly, but whether distillation beats from-scratch training in absolute terms depends on the teacher–student capacity gap.

B. Limitations

Several limitations bound the present work:

- Abbreviated training schedules: FP32 baselines (78–79% MobileNetV2 over 15 epochs, 66.8% EfficientNet-B0 over 15 epochs) sit below published state-of-the-art ($\sim$94% for both with 200+-epoch schedules and stronger augmentation). Relative compression and accuracy-degradation metrics remain valid in this regime, but absolute differences at higher baselines may shift.
- Three-seed reporting: The revised study reports mean$\pm$std over three random seeds; this captures first-order training stochasticity but is too few seeds for tight 95% confidence intervals on small effects. Five-to-ten seeds would be preferable and is on the future-work shelf.
- x86-64 CPU as the on-device proxy: The latency measurement uses an Intel Xeon-class server CPU with FBGEMM; this is a representative deployment-class CPU but is not a Cortex-M microcontroller or a Raspberry Pi. The repository ships an `edge_latency.py` that runs unchanged on aarch64/armv7 hardware, but the in-paper numbers are not from such hardware.
- Dataset and architecture scope: The benchmark suite covers CIFAR-10 and Fashion-MNIST with five architectures. Extension to ImageNet-scale models and to transformer architectures is left to users and is the third future-work item.
- Wrapper-level contribution: Malak wraps existing PyTorch primitives rather than introducing new compression algorithms. Its value lies in reducing boilerplate and improving reproducibility, not in algorithmic novelty.
- Embedded deployment is proof-of-concept: The Cortex-M7 C inference code uses naive reference kernels and serves to demonstrate the export pipeline end-to-end; production deployment should terminate this path in CMSIS-NN.

C. Comparison with Existing Tools, Revisited

Malak's primary advantage over raw PyTorch is the reduction in boilerplate: the QAT workflow shrinks from $\sim$40 lines to the five-line example in Section III, and the full `train`→`compress`→`export`→`profile` loop fits on a single screen. This advantage is real for the exploratory and teaching settings the toolkit targets, and uninteresting for production pipelines where the boilerplate is written once and amortized.

Against Intel Neural Compressor [9], Malak trades a strictly smaller compression menu and no hardware-aware tuning for a simpler dependency footprint and ARM-friendly defaults.

Against TFLite [5], Malak stays inside the PyTorch ecosystem at the cost of a production-grade embedded runtime. The toolkit is therefore positioned as a starting point, not an end point: users whose work outgrows it should expect to graduate to one of these heavier tools, and the export pipeline (ONNX, INT8 weight files) is designed to make that transition cheap.

VI. Conclusion

We presented Malak, an open-source Python toolkit that wraps PyTorch's compression primitives in a thin, task-oriented API aimed at research prototyping and graduate-level instruction, where the standard 40–60-line QAT boilerplate is disproportionately costly. Six experiments validate the wrapped pipelines. QAT delivers $3.48\times$ compression with no statistically meaningful accuracy change on MobileNetV2, and magnitude pruning at 50% sparsity holds within $\sim$ 1 pp of baseline. The `KnowledgeDistiller` module produces the qualitative effect Hinton-style soft-label transfer predicts; the gap's sign depends on the teacher-student capacity gap. INT8 static quantization yields a $2.16\times$ wall-clock speedup over FP32 on a deployment-class CPU, and the same SimpleCNN INT8 binary fits and runs on a simulated STM32H7 (Cortex-M7) under Renode.

Malak does not claim to be a production deployment runtime: the C inference path is a proof-of-concept that should terminate in CMSIS-NN before being trusted in the field. It is also not a competitor to Intel Neural Compressor's strategy library, which offers a strictly larger compression menu and remains the right choice when its hardware assumptions match the deployment target. The contribution is engineering and pedagogical rather than algorithmic: the toolkit's value lies in shrinking the activation energy required to start a compression study within the PyTorch ecosystem, not in introducing new compression methods.

Future work, ordered by practical leverage:

- Production-grade embedded backend: Replace the proof-of-concept ARM Cortex-M7 C kernels with a CMSIS-NN integration [8] so the Malak export path terminates in optimized, INT8-correct inference rather than naive reference code. This is the single most-requested missing piece for users with hardware in hand.
- Mixed-precision and hardware-aware quantization: Add support for HAQ-style mixed-precision [15] and per-target qconfigs, since uniform INT8 leaves measurable accuracy on the table at the same compression ratio.

- Larger-scale validation: Extend the benchmark suite to ImageNet-scale models and at least one transformer architecture (e.g., DeiT-Tiny) to probe whether the wrapper holds up beyond convolutional networks.
- Automated compression policy search: Combine quantization, pruning, and distillation under a single search loop, building on the existing per-module APIs.
- Continuous statistical reporting: The current study reports three seeds; the longer-term aim is for every published Malak result to ship with confidence intervals by default.

The toolkit is available at https://github.com/alnemari-m/malak-platform under the MIT license.

ACKNOWLEDGMENT

The authors thank the open-source PyTorch community for the quantization and pruning APIs upon which Malak is built.

REFERENCES

[1] P. Warden and D. Situnayake, *TinyML: Machine Learning with TensorFlow Lite on Arduino and Ultra-Low-Power Microcontrollers*. O'Reilly Media, 2019.

[2] Z. Zhou, X. Chen, E. Li, L. Zeng, K. Luo, and J. Zhang, "Edge intelligence: Paving the last mile of artificial intelligence with edge computing," *Proc. IEEE*, vol. 107, no. 8, pp. 1738–1762, 2019.

[3] B. Jacob *et al.*, "Quantization and training of neural networks for efficient integer-arithmetic-only inference," in *Proc. IEEE CVPR*, 2018, pp. 2704–2713.

[4] S. K. Esser, J. L. McKinstry, D. Bablani, R. Appuswamy, and D. S. Modha, "Learned step size quantization," in *Proc. Int. Conf. Learn. Represent. (ICLR)*, 2020.

[5] TensorFlow Authors, "TensorFlow Lite: Deploy machine learning models on mobile and edge devices," Documentation, Google LLC, 2024. [Online]. Available: https://www.tensorflow.org/lite. Accessed: April 2026.

[6] R. David, J. Duke, A. Jain, V. Janapa Reddi, N. Jeffries, J. Li, N. Kreeger, I. Nappier, M. Natraj, T. Wang *et al.*, "TensorFlow Lite Micro: Embedded machine learning for TinyML systems," in *Proc. MLSys*, vol. 3, 2021, pp. 800–811.

[7] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, "TVM: An automated end-to-end optimizing compiler for deep learning," in *Proc. USENIX OSDI*, 2018, pp. 578–594.

[8] L. Lai, N. Suda, and V. Chandra, "CMSIS-NN: Efficient neural network kernels for Arm Cortex-M CPUs," *arXiv preprint* arXiv:1801.06601, 2018. [Online]. Available: https://arxiv.org/abs/1801.06601.

[9] F. Tian, H. Shen *et al.*, "Intel® Neural Compressor: An open-source Python library for model compression," Intel Corporation, 2024. [Online]. Available: https://github.com/intel/neural-compressor. Accessed: April 2026.

[10] L. Deng, G. Li, S. Han, L. Shi, and Y. Xie, "Model compression and hardware acceleration for neural networks: A comprehensive survey," *Proc. IEEE*, vol. 108, no. 4, pp. 485–532, 2020.

[11] G. Hinton, O. Vinyals, and J. Dean, "Distilling the knowledge in a neural network," in *NIPS Deep Learning Workshop*, 2015.

[12] Z. Liu, J. Li, Z. Shen, G. Huang, S. Yan, and C. Zhang, "Learning efficient convolutional networks through network slimming," in *Proc. IEEE Int. Conf. Comput. Vis. (ICCV)*, 2017, pp. 2755–2763.

[13] J. Frankle and M. Carbin, "The lottery ticket hypothesis: Finding sparse, trainable neural networks," in *Proc. Int. Conf. Learn. Represent. (ICLR)*, 2019.

[14] M. Tan and Q. Le, "EfficientNet: Rethinking model scaling for convolutional neural networks," in *Proc. ICML*, 2019, pp. 6105–6114.

[15] K. Wang, Z. Liu, Y. Lin, J. Lin, and S. Han, "HAQ: Hardware-aware automated quantization with mixed precision," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR)*, 2019, pp. 8604–8612.

[16] J. Bai, F. Lu, K. Zhang *et al.*, "ONNX: Open Neural Network Exchange," Linux Foundation AI & Data, 2019. [Online]. Available: https://github.com/onnx/onnx. Accessed: April 2026.

[17] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "MobileNetV2: Inverted residuals and linear bottlenecks," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR)*, 2018, pp. 4510–4520.

[18] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, 2016, pp. 770–778.

[19] PyTorch Team, "Quantization — PyTorch 2.x documentation," Meta Platforms, 2024. [Online]. Available: https://pytorch.org/docs/stable/quantization.html. Accessed: April 2026.

[20] D. Sculley, G. Holt, D. Golovin, E. Davydov, T. Phillips, D. Ebner, V. Chaudhary, M. Young, J.-F. Crespo, and D. Dennison, "Hidden technical debt in machine learning systems," in *Adv. Neural Inf. Process. Syst. (NeurIPS)*, vol. 28, 2015, pp. 2503–2511.

[21] M. Zaharia, A. Chen, A. Davidson, A. Ghodsi, S. A. Hong, A. Konwinski, S. Murching, T. Nykodym, P. Ogilvie, M. Parkhe, F. Xie, and C. Zumar, "Accelerating the machine learning lifecycle with MLflow," *IEEE Data Eng. Bull.*, vol. 41, no. 4, pp. 39–45, 2018.

[22] J. Pineau, P. Vincent-Lamarre, K. Sinha, V. Larivière, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and H. Larochelle, "Improving reproducibility in machine learning research: A report from the NeurIPS 2019 reproducibility program," *J. Mach. Learn. Res.*, vol. 22, no. 164, pp. 1–20, 2021.

[23] Antmicro, "Renode: Open-source virtual development framework for multi-node embedded systems," Antmicro Ltd., 2024. [Online]. Available: https://renode.io. Accessed: April 2026.