Multi-Stage Compression of Machine Learning Models

Nahyeon Kim and Prabhat Mishra
University of Florida, Gainesville, Florida, USA

***Abstract*—Deploying machine learning (ML) models on edge devices is challenging due to strict memory and computational constraints. To mitigate these limitations, various model compression techniques, such as pruning, tensor decomposition, quantization, and Huffman coding, have been explored. However, most prior efforts perform compression on trained ML models, which requires significant fine-tuning to recover accuracy. In this work, we employ a multi-stage compression strategy that effectively combines pre-training pruning and tensor decomposition with post-training quantization and Huffman coding. The pre-training compression eliminates the need for costly fine-tuning for preserving model accuracy. It also reduces both training and inference memory requirements. To reduce the memory requirements further, we perform post-training quantization and Huffman coding. Unlike conventional quantization approaches that use a single scale factor per layer, we assign different scale factors to each decomposed core, thereby minimizing information loss. Experimental evaluation demonstrates that our approach significantly reduces the model size (145× for ResNet101, 114× for ResNet50) with minor accuracy loss.**

***Index Terms*—Machine learning, model compression, tensor decomposition, quantization, Huffman coding**

I. Introduction

Modern machine learning models have grown rapidly in both capacity and complexity. As model accuracy improves, so do the demands on memory, energy consumption, and storage. This growth in resource requirements makes it increasingly difficult to deploy high-performance models on resource-constrained edge devices. At the same time, the number and variety of edge platforms that could benefit from on-device intelligence are proliferating, which amplifies the urgency of developing compact, energy-efficient models that retain strong predictive performance.

A. State-of-the-Art and Limitations

To address this challenge, a wide range of model compression techniques have been proposed, including pruning, tensor decomposition, quantization, and Huffman coding [1]–[9]. Recent works frequently combine two or more of these approaches to reduce model size and memory footprint. However, there is no consensus on the optimal combination or ordering of compression stages, and different pipelines often yield different trade-offs among compression ratio, accuracy degradation, and the need for additional fine-tuning. Moreover, most existing works focus on post-training compression, overlooking the energy consumption incurred during training. These gaps motivate the need for a systematic investigation of multi-stage compression strategies, particularly those that operate in a pre-training compression framework.

B. Contributions

In this work, we introduce **QuadPress**, a multi-stage compression framework that explicitly integrates pre-training structural reduction (pruning and tensor decomposition) with post-training numerical reduction (quantization and Huffman coding). Specifically, we first impose structured pruning on the untrained network to produce a sparse architecture and then apply tensor decomposition to the pruned weights to further reduce parameter count and intermediate storage. The compressed architecture is trained from scratch in its reduced form, which avoids the heavy fine-tuning typically required when compressing a fully trained model. After training, we perform mixed-scale quantization on the decomposed tensor cores—assigning distinct scale factors to each core rather than using a single layer-wise scale. Finally, we apply Huffman coding to the quantized integer weights to achieve storage savings.

We have evaluated the effectiveness of our framework using two popular convolutional neural networks (ResNet50 and ResNet101 [10]). Our experimental results demonstrate that the proposed pipeline attains substantially higher memory reduction with minimal accuracy loss when compared to state-of-the-art methods. Specifically, this paper makes the following major contributions:

1) We propose a pre-training compression strategy that combines structured pruning and tensor decomposition to reduce training-time memory and compute requirements.

2) We demonstrate that pre-training compression eliminates the need for costly post-compression fine-tuning, simplifying deployment workflows.
3) We employ a mixed-scale quantization scheme at the level of decomposed tensor cores, which minimizes information loss compared to single-scale quantization, and combine it with Huffman coding for further model size reduction.

The rest of the paper is organized as follows. Section II reviews the related literature on compression pipelines. Section III details our multi-stage compression pipeline and implementation choices. Section IV presents experimental results and ablations. Finally, Section V concludes the paper.

II. Background and Related Work

We first provide an overview of four compression methods: pruning, tensor decomposition, quantization, and Huffman coding. Next, we survey related efforts that utilize a combination of these methods.

A. Background: Major Compression Methods

1) Pruning: Pruning tries to remove less important weights in a neural network, reducing the number of parameters and computational cost while minimally affecting accuracy [11]. Early approaches analyze the sensitivity of the loss to individual weights using the Hessian matrix, enabling selective removal of low-importance parameters. Structured pruning further removes entire neurons, filters, or channels, making the model more hardware-friendly.

2) Tensor Decomposition: Tensor decomposition replaces dense weight tensors with low-rank or factorized representations to reduce parameter count and computational complexity [3]. Common decomposition methods include CP, Tucker, and Tensor-Train, which approximate high-dimensional weight tensors as a product of smaller factor matrices or cores. This approach preserves the network's expressiveness while compressing it efficiently.

3) Quantization: Quantization maps high-precision floating-point weights (and sometimes activations) to low-bit discrete representations such as int8, int4, or even binary [2]. This reduces memory usage and enables efficient integer-arithmetic inference on resource-constrained hardware. Both uniform and non-uniform quantization schemes exist, with some methods learning optimal codebooks for better accuracy.

4) Huffman Coding: Huffman coding is an entropy coding method that losslessly compresses discrete symbols by assigning shorter codes to more frequent symbols and longer codes to rare ones [12]. When applied to quantized network weights or indices, it reduces storage requirements without impacting accuracy. Huffman coding is particularly effective when combined with pruning and quantization, exploiting the non-uniform distribution of weight values. The method was first applied to compressed neural networks in the deep compression pipeline [1].

B. Related Work: Hybrid Approaches

Researchers have explored various combinations of compression methods that can be broadly divided into three categories: (1) pruning combined with tensor decomposition (PR+TD), where low-rank factorization and structural sparsification are used together to reduce both parameters and runtime; (2) pruning, quantization, and Huffman coding (PR+QU+HC) that maximize final storage reduction for deployment; and (3) tensor-decomposition with quantization (TD+QU), which couples model compact parameterization with low-precision encoding.

Recent efforts in PR+TD search for complementary compression modes per-layer and jointly optimize rates across layers to preserve accuracy while achieving high compression ratios [13], [14]. In case of PR+QU+HC, the canonical deep compression pipeline and follow-up extensions apply pruning, then trained quantization, and finally Huffman coding to obtain very small stored models. The frameworks that performs pruning and quantization in parallel attempt to learn sparsity and quantized weights jointly to avoid separate stages [1], [15]. For TD+QU, QuaDCNN applies tensor train decomposition together with quantization-aware rank/bit choices to achieve compact CNNs [16]. Other studies investigate additional compression gains by exploiting repeated patterns caused by pruning/quantization and applying advanced entropy schemes beyond Huffman coding [17].

Despite strong compression ratios at inference time, an important limitation of most existing hybrid approaches is that they are *post-training*. Post-training pipelines do not reduce the memory or energy required during the initial full-precision training phase, which remains a dominant cost for large models and for scenarios requiring on-device or continual learning [18]. Moreover, aggressive post-training compression often demands extended fine-tuning to recover pre-compression accuracy—an imprac-

tical overhead when model updates must be frequent or when training resources are constrained. These gaps motivate the need for unified approaches that can enable compression earlier (pre-training or during training) and combine structured, mixed-granularity techniques (structured pruning, tensor decomposition, per-core/mixed-scale quantization, followed by entropy coding) to reduce both training-time resource use and the final deployed model size. This paper tries to fulfill this need.

III. Multi-Stage Compression

Figure 1 shows our proposed framework, **Quad-Press**, that effectively combines pre-training optimization (pruning and tensor decomposition) with the post-training compression (quantization and Huffman coding). Specifically, our framework consists of four major steps: pruning, tensor train decomposition, quantization, and Huffman coding. The rest of this section describes these steps.

A. Pre-training Pruning

Structured channel pruning is applied to reduce the network capacity at the architectural level prior to any training. By reducing convolutional output channels (filters), the method enforces sparsity that directly translates into fewer parameters, reduced memory footprint, and lower arithmetic cost during both training and inference. The pruning process is implemented using the `torch_pruning` [4] library with the `MagnitudePruner` API.

Channels are scored by a simple, data-free importance measure based on the magnitude of their weights: each channel's importance is computed as the L_2-norm of its convolutional kernel, and channels with the smallest norms are selected for removal. A global target sparsity r (the fraction of channels to remove) is applied across eligible convolutional layers. However, certain layers that are sensitive to structural changes (e.g., the final classifier) are exempted from pruning. To guarantee consistency of feature-map shapes after deletion, pruning is performed in a dependency-aware manner so that correlated tensors and their consumers are updated coherently (e.g., BatchNorm layers), avoiding shape mismatches and preserving network connectivity.

In our framework, pruning is applied prior to model training (pre-training pruning) with a sparsity of 0.8 for both ResNet101 and ResNet50. After tensor decomposition, the pruned architecture is trained from scratch, allowing the benefits of reduced peak memory usage and improved training throughput to be realized throughout the optimization process.

B. Tensor Decomposition

After structured pruning, we apply Tensor-Train (TT) decomposition to further compress selected convolutional kernels, leveraging the complementary relationship between TT factorization and structured pruning. TT reduces the dimensionality of convolutional kernels by representing them with compact core tensors, removing redundant directions in the weight space and alleviating the need for aggressive channel pruning. Conversely, lowering TT-ranks acts as a structured analogue of pruning, eliminating entire degrees of freedom rather than individual weights. Additionally, TT-cores expose modes corresponding to input/output channel groups, allowing structured sparsity to be imposed directly in the factorized domain. Because TT-cores shrink along pruned channel dimensions, the decomposition naturally adapts to architectures modified by dependency-aware pruning. For simplicity and stable rank selection, we first perform pruning and then replace the remaining large convolutional layers with their TT-decomposed counterparts.

Decomposition is restricted to convolutional layers in our framework because they typically dominate model parameter counts and FLOPs. The weight tensor of each convolutional layer can be decomposed with a specified rank value. For example, Conv2d layer's weights are shaped as (I, k_H, k_W, O), and when it is decomposed by tensor train using rank $(1, r_1, r_2, r_3, 1)$, the final decomposed weight becomes $(1, I, r_1), (r_1, k_H, r_2), (r_2, k_W, r_3), (r_3, O, 1)$. This requires the calculation involving the weight tensor to be updated to operate using the decomposed factors of the decomposed tensor instead of the single, original tensor.

Convolutional operation over decomposed weights follows this flow: contraction with each core starting from the core that contains I (input channels), intermediate cores that contain the kernel size, and finally the core that contains O (output channels). In case of convolution over TT-cores, let

$$X \in \mathbb{R}^{B\times I\times H\times W}, \qquad \mathcal{G}^{(1)} \in \mathbb{R}^{1\times I\times r_1}.$$

First, we perform a 1×1 convolution with $\mathcal{G}^{(1)}$:

$$X_1 = \mathrm{Conv1d}(X, \mathcal{G}^{(1)}) \in \mathbb{R}^{B\times r_1\times H\times W},$$

Next, using the two intermediate TT cores

$$\mathcal{G}^{(2)} \in \mathbb{R}^{r_1\times K_H\times r_2}, \quad \mathcal{G}^{(3)} \in \mathbb{R}^{r_2\times K_W\times r_3},$$

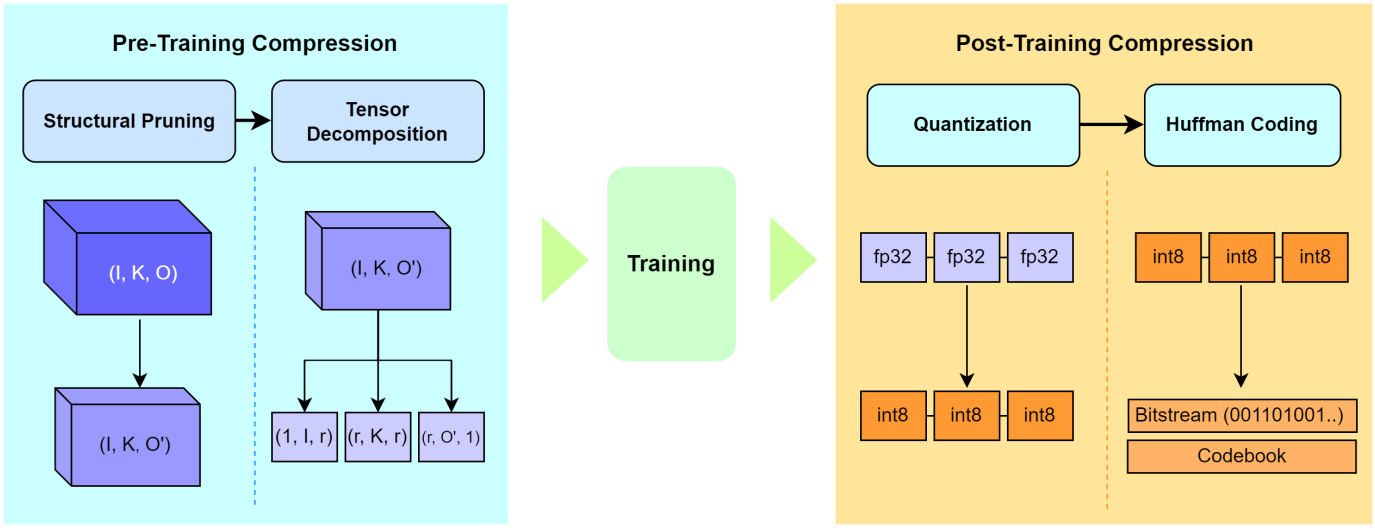


Fig. 1: Overview of the QuadPress compression pipeline that consists of four major stages: pruning, tensor decomposition, quantization, and Huffman coding.

we apply consecutive 2D convolutions:

$$X_2 = \text{Conv2d}(X_1, \mathcal{G}^{(2)}) \in \mathbb{R}^{B\times r_2\times H'\times W'}$$

$$X_3 = \text{Conv2d}(X_2, \mathcal{G}^{(3)}) \in \mathbb{R}^{B\times r_3\times H''\times W''}.$$

where H', H'', W' and W'' depend on stride and padding. Finally, with the last core

$$\mathcal{G}^{(4)} \in \mathbb{R}^{r_3\times O\times 1},$$

we perform another 1×1 convolution:

$$Y = \text{Conv1d}(X_3, \mathcal{G}^{(4)}) \in \mathbb{R}^{B\times O\times H''\times W''}.$$

In this way, each TT-core is contracted sequentially without reconstructing the full weight tensor. It also allows the model to be trained while preserving the pruned and decomposed architecture.

C. Post-training Quantization

We adopt a post-training symmetric integer quantization scheme targeted at models that have been structurally pruned and converted to use tensor-decomposed convolutional kernels. The quantization procedure is designed to be simple, reproducible, and to balance compression with minimal accuracy degradation. While the trained weights are originally in float32 format, converting them to int8 allows for a substantial reduction in memory usage. In practice, this requires both deciding how the scale should be computed and specifying layer-specific quantization rules.

1) Scale computation.: For a floating-point weight tensor W, we compute a symmetric linear scale s that maps floating values to a signed b-bit integer range. Let $Q = 2^{b-1} - 1$. Two complementary strategies for scale estimation are used:

- **Per-tensor / per-channel maximum:** $s = \frac{\max(|W|)}{Q}$ for per-tensor quantization, or compute $\max(|W|)$ along the quantization axis (e.g., output channels) to obtain a vector of per-channel scales.
- **Percentile-robust scale:** To reduce sensitivity to rare large outliers, the scale is optionally based on a high percentile of the absolute value distribution, i.e., $s = \frac{\text{quantile}(|W|,p)}{Q}$ for percentile p (e.g., $p \approx 99.9\%$).

Per-channel scales are recommended for convolutional and fully connected weights; per-tensor scales are used for small 1-D parameters and non-weight buffers. With scale s and integer bound Q, quantization is performed as

$$q = \text{clip}\big(\text{round}(W/s),\, -Q,\, Q\big)$$

2) Layer-specific rules.: Different parameter types follow different quantization granularities:

- **Convolutional kernels:** quantized per output channel (per-row in flattened view). This preserves relative dynamic range between channels and typically reduces accuracy loss.
- **Linear / fully connected weights:** quantized per output row (analogous to per-channel).
- **Tensor-train (TT) cores:** two operational modes are supported: (i) a *fixed* global scale shared across

all TT cores, and (ii) an *adjusted* mode where each core receives its own scale (percentile-based). The adjusted mode yields finer fidelity at the cost of slightly more metadata.

- **Biases / small buffers:** quantized with a per-tensor scheme.

These strategies allow the quantization to be consistent and stable across different parameter/layer types.

D. Huffman Coding

After pruning, tensor-train decomposition and int8 quantization, we apply a lossless entropy coder (Huffman coding) to further reduce the serialized model size without introducing any additional approximation error. The Huffman compression is performed on the raw int8 payload of the quantized state dictionary.

Specifically, all `int8` tensors in the quantized `state_dict` are extracted and converted to their raw byte representations. The byte streams are concatenated in a deterministic key order, and a frequency table of byte values (0–255) is computed. We build a Huffman tree from this frequency table using a min-heap (priority queue), producing a prefix-free binary code (codebook) that maps each byte symbol to a variable-length bit-string. The concatenated byte stream is encoded by substituting each byte with its corresponding code and packing the resulting bitstream MSB-first into bytes. The encoder also records the number of valid bits in the final output byte to allow exact reconstruction.

The compression package contains: (i) the packed Huffman-encoded bytes, (ii) the Huffman codebook, (iii) the deterministic ordering and shapes of the `int8` tensors, and (iv) all non-`int8` metadata. This package is serialized (we use Python pickle in the implementation) and its file size is reported as the compressed model size.

IV. Experiments

We have evaluated the effectiveness of our multi-stage compression framework. We first outline the experimental setup. Next, we present the results.

A. Experimental Setup

All experiments are performed on an NVIDIA B200 GPU using two representative convolutional architectures: ResNet50 and ResNet101. We have evaluated our compression pipeline on the CIFAR-10 dataset [19], a standard benchmark containing 60,000 images across 10 classes. Benchmark studies for ResNet50 typically use ImageNet [20]; however, due to resource constraints, we conducted our experiments on CIFAR10. Therefore, comparisons are made based on relative accuracy loss rather than absolute accuracy values. To quantify compression effectiveness, we report FLOPs (Floating Point Operations), parameter count, file storage size, and accuracy drop. FLOPs represent the total number of multiply-add operations a model performs, and accuracy drop is computed as the difference between the original and compressed model accuracies. We also report the reductions in FLOPs, parameter count, and file size relative to the uncompressed baseline, calculated as the ratio of the original value to the compressed value (e.g., $\text{FLOPs}_{original}/\text{FLOPs}_{compressed}$). The baseline model results are presented in Table I. Training and inference energy consumption are measured using the `pynvml` library, where GPU power is sampled every 0.1 seconds and per-epoch energy is calculated as $\sum(0.1 \times W)$ over the entire epoch duration, providing a practical estimate of total energy usage.

TABLE I: The FLOPs, number of parameters, model file size, and accuracy of baseline networks.

	FLOPs (G)	**Param (M)**	**Size (MB)**	**Acc (%)**
ResNet101	15.73	42.513	162.782	90.86
ResNet50	8.265	25.557	90.033	91.26

All models are trained for 150 epochs using a batch size of 128. We use an initial learning rate of 0.01 with a patience-based decay policy, where the learning rate is reduced if validation performance does not improve for 5 consecutive evaluations. Weight decay is set to 5×10^{-4} for regularization.

B. Results

Figure 2 illustrates why pre-training compression is more effective than post-training compression. We plot the validation accuracy curves of two settings: (1) training a model after applying pruning and tensor-train decomposition, and (2) applying pruning and tensor-train decomposition to a fully trained model followed by fine-tuning. As shown in the figure, post-training compression causes a significant accuracy drop immediately after compression, requiring almost the same number of training epochs as the original training to recover the baseline accuracy. In contrast, our pre-training compression approach achieves high accuracy within a similar number of epochs as the original training, without requiring any fine-tuning.

Specifically, both configurations were initially trained for 60 epochs. The pre-training configuration had pruning+TD applied prior to optimization and was trained

TABLE II: Compression results for ResNet101 and ResNet50 under various methods. FLOPs is number of floating-point operations in billions, and Param is parameter count in millions. × indicates the improvement (how many times smaller it is compared to the uncompressed model). Size indicates the model file storage size in MB. ΔAcc (%) indicates the accuracy drop, computed as the difference between the original and compressed model accuracies.

Model	Compression Technique	FLOPs	FLOPs (×)	Param	Param (×)	Size	Size (×)	ΔAcc (%)
ResNet101	Uncompressed ResNet101	15.73	1	42.51	1	162.78	1	0.0
	Only pruning	0.66	23.83	2.11	20.15	6.79	23.98	1.79
	Only Tensor Train	3.71	4.24	6.59	6.45	18.07	9.00	-0.14
	Only Quantization	15.73	1	42.51	1	41.74	3.9	-0.04
	PR + QU	0.66	23.83	2.11	20.15	2.08	78.26	1.41
	TD + QU (fixed scale)	3.71	4.24	6.59	6.45	5.64	28.89	0.13
	TD + QU (mixed scale)	3.71	4.24	6.59	6.45	5.63	28.92	-0.11
	PR + TD	0.19	82.79	1.28	33.21	3.67	44.32	2.19
	PR + TD + QU (fixed scale)	0.19	82.79	1.28	33.21	1.47	110.73	2.35
	PR + TD + QU (mixed scale)	0.19	82.79	1.28	33.21	1.47	110.73	2.05
	QuadPress	0.19	82.79	1.28	33.21	1.12	145.34	2.05
	QuaDCNN (TD + QU) [16]	8.43	1.87	22.86	1.86	89.52	1.82	0.87
ResNet50	Uncompressed ResNet50	8.27	1	25.56	1	90.03	1	0.0
	Only pruning	0.36	22.96	1.35	18.93	3.74	23.09	3.01
	Only Tensor Train	6.62	1.25	11.96	2.14	38.17	2.36	1.13
	Only Quantization	8.27	1	25.56	1	23.03	3.91	0.03
	PR + QU	0.36	22.96	1.35	18.93	1.13	79.68	3.01
	TD + QU (fixed scale)	6.62	1.25	11.96	2.14	10.04	8.96	1.19
	TD + QU (mixed scale)	6.62	1.25	11.96	2.14	10.04	8.96	1.11
	PR + TD	0.30	27.55	1.12	22.82	2.87	31.40	1.78
	PR + TD + QU (fixed scale)	0.30	27.55	1.12	22.82	0.95	94.77	1.82
	PR + TD + QU (mixed scale)	0.30	27.55	1.12	22.82	0.95	94.77	1.72
	QuadPress	0.30	27.55	1.12	22.82	0.79	114.26	1.72
	NORTON (PR + TD) [14]	4.16	1.99	13.51	1.89	-	-	-0.43
	CLIP-Q (PR + QU) [15]	-	-	-	-	6.7	13.44	-0.6
	QuaDCNN (TD + QU) [16]	4.64	1.78	13.39	1.91	52.75	1.71	0.41

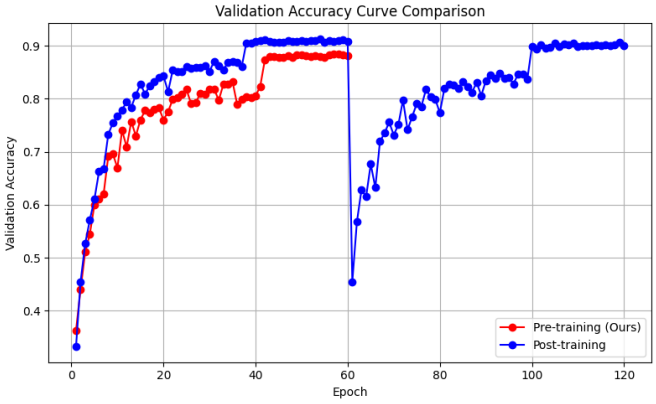
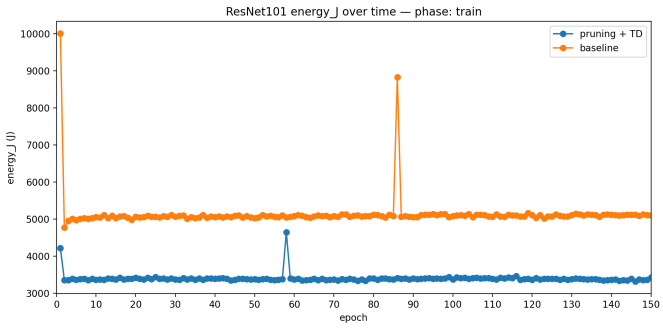


Fig. 2: Pre-training vs post-training pruning and tensor decomposition. Our approach does not require expensive fine-tuning.

for 60 epochs starting from the compressed initialization. The post-training configuration was trained in the standard (uncompressed) form for 60 epochs, after which pruning+TD were applied and the resulting compressed model was fine-tuned for an additional 60 epochs. Consequently, the post-training workflow required a total of 120 epochs of optimization (60 initial + 60 fine-tuning) to approach the baseline accuracy, whereas the pre-trained compressed model reached comparable accuracy within the original 60 epochs and without any subsequent fine-tuning.

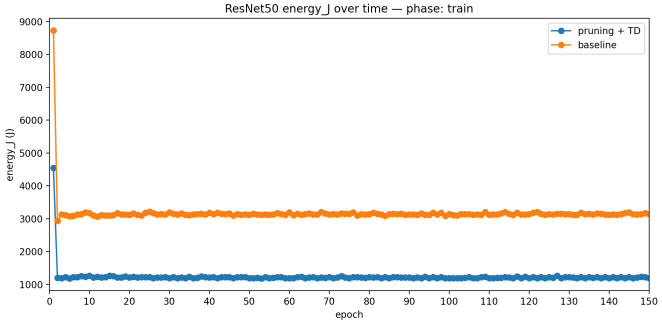
We measured the energy consumption per epoch of ResNet101 and ResNet50 models running on a GPU. Figure 3a and 3c shows the comparison of energy consumption during training between the original baseline model and the compressed model, while Figure 3b and 3d shows the comparison during inference. As illustrated, energy consumption decreased for both training and inference in both models. ResNet101 exhibited a 1.5× reduction in energy for both training and inference, whereas ResNet50 achieved reductions of 2.6× during training and 2.5× during inference. These results demonstrate that our pre-training compression approach effectively reduces the overall energy required for both training and inference.

Table II presents an overall comparison between our proposed approach (QuadPress) and state-of-the-art [14]–[16] for ResNet101 and ResNet50. The first entry in each category represents the original (uncompressed) model. Note that our QuadPress utilizes four

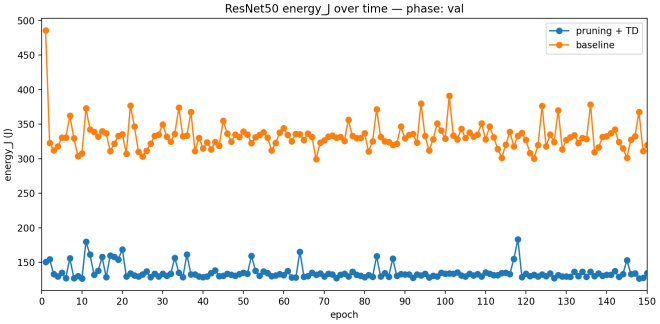


(a) Energy consumption during training for ResNet101

(b) Energy consumption during inference for ResNet101



(c) Energy consumption during training for ResNet50



(d) Energy consumption during inference for ResNet50

Fig. 3: Comparison of energy consumption during training and inference. Our framework (pruning+TD) significantly outperforms (1.5× reduction in energy for both training and inference for ResNet101, and reductions of 2.6× and 2.5× for training and inference, respectively, for ResNet50.) compared to the baseline (Table I).

components: pruning (PR), tensor decomposition (TD), mixed scale quantization (QU), and Huffman coding (HC). The table shows how combinations of these components contributes to the overall savings.

Overall, our method achieves lower FLOPs, fewer parameters, and smaller file sizes compared to the benchmarks. For ResNet101, pruning alone significantly reduces FLOPs (0.66 G, 23.83×) and parameters (2.11 M, 20.15×) but incurs an accuracy loss of 1.79%. Tensor-train (TT) decomposition preserves accuracy (-0.14%) but yields only moderate reductions in computation and storage, while quantization alone has negligible impact on accuracy (-0.04%) and only modestly reduces on-disk size. Combining pruning with tensor-train decomposition (PR+TD) produces a strong synergy, lowering FLOPs to 0.19 G (82.79×) and parameters to 1.28 M (33.21×) at a modest accuracy degradation (2.19%). When applying quantization, we observe that mixed-scale quantization retains accuracy 0.3% better than fixed-scale quantization, indicating better information preservation. Adding quantization (fixed or mixed scale) further reduces the stored model size to $\approx$ 1.47 MB (110×) while keeping accuracy loss small. Finally, our QuadPress that integrates Huffman coding on top of PR+TD+mixed-scale quantization achieves the smallest file size of 1.120 MB (145×) with only 2.05% accuracy degradation, outperforming the QuaDCNN benchmark [16] in terms of computational and storage efficiency.

For ResNet50, a similar trend is observed. Pruning alone reduces FLOPs (0.36 G, 22.96×) and parameters (1.35 M, 18.931×) but at the cost of a higher accuracy loss (3.01%). Tensor-train decomposition and quantization individually offer limited benefits for computation and storage (TD and QU show only modest or mixed improvements), while their combination (TD+QU) reduces on-disk size (to $\approx$ 10.04 MB, 8.96×) with moderate accuracy degradation (1.11–1.19%). Our proposed pipeline QuadPress reduces FLOPs to 0.30 G (27.55×), parameters to 1.12 M (22.82×), and file size to 0.788 MB (114×), while maintaining an accuracy loss of only 1.72%. Because Huffman coding is a lossless entropy-coding scheme, it does not introduce additional accuracy degradation; this is reflected in Table II, where the reported ΔAcc remains unchanged after applying HC on top of PR+TD+QU (mixed scale).

It is important to note that post-training techniques such as quantization and Huffman coding primarily reduce on-disk model size but do not change the model architecture or the number of arithmetic operations. Therefore, they do not reduce FLOPs or parameter counts. Pre-training pruning combined with tensor decomposition, on the other hand, alters the model representation and provides the necessary reductions in FLOPs and parameter count, which complements the storage savings achieved by post-training methods. Our experimental results support this behavior. For example, in Table II the quantization-only ResNet101 entry shows a large on-disk reduction (41.74 MB) while retaining high computational cost (15.73 G FLOPs, 42.51 M params), whereas PR+TD reduces FLOPs and parameters dramatically (0.19 G FLOPs, 1.28 M params) while also enabling much smaller stored models. A similar pattern is observed for ResNet50 (quantization-only: 8.27 G FLOPs and 23.03 MB on disk versus PR+TD: 0.30 G FLOPs and 2.87 MB on disk), confirming that pre-training structural compression is necessary when reductions in runtime computation and parameter count are desired.

Compared to existing methods such as NORTON [14], CLIP-Q [15], and QuaDCNN [16], our approach consistently achieves substantially lower computational and storage costs while incurring only a minor accuracy penalty. These results demonstrate that integrating pre-training pruning, tensor-train decomposition, mixed-scale quantization, and lossless entropy coding yields a favorable trade-off between compression and predictive performance.

V. Conclusion

Designing energy-efficient machine learning pipeline needs to ensure energy-efficiency during both training and inference. While there are promising approaches for compressing trained machine learning models, they miss pre-training optimization opportunities. Moreover, they may require costly fine-tuning to recover accuracy. In this paper, we propose QuadPress that combines the advantages of pre-training pruning and tensor decomposition with post-training quantization and Huffman coding. The pre-training compression reduces both training and inference memory requirements. It also eliminates the need for costly fine-tuning for preserving model accuracy. Extensive experimental evaluation using two convolutional architectures with CIFAR-10 dataset demonstrated that our approach can significantly reduces the model size (145× for ResNet101, 114× for ResNet50) with minor accuracy loss.

References

[1] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *arXiv preprint arXiv:1510.00149*, 2015.

[2] Y. Gong and e. a. Liu, "Compressing deep convolutional networks using vector quantization," 2014.

[3] M. Jaderberg, A. Vedaldi, and A. Zisserman, "Speeding up convolutional neural networks with low rank expansions," *arXiv preprint arXiv:1405.3866*, 2014.

[4] G. Fang, X. Ma, M. Song, M. B. Mi, and X. Wang, "Depgraph: Towards any structural pruning," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2023, pp. 16 091–16 101.

[5] A. Novikov, D. Podoprikhin, A. Osokin, and D. P. Vetrov, "Tensorizing neural networks," *Advances in neural information processing systems*, vol. 28, 2015.

[6] B. Jacob and e. a. Sklyarov, "Quantization and training of neural networks for efficient integer-arithmetic-only inference," 2018.

[7] V. Lebedev, Y. Ganin, M. Rakhuba, I. Oseledets, and V. Lempitsky, "Speeding-up convolutional neural networks using finetuned cp-decomposition," *arXiv:1412.6553*, 2014.

[8] N. Lee, T. Ajanthan, and P. H. Torr, "Snip: Single-shot network pruning based on connection sensitivity," *arXiv preprint arXiv:1810.02340*, 2018.

[9] C. Wang, G. Zhang, and R. Grosse, "Picking winning tickets before training by preserving gradient flow," *arXiv preprint arXiv:2002.07376*, 2020.

[10] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[11] Y. LeCun, J. Denker, and S. Solla, "Optimal brain damage," *Advances in neural information processing systems*, 1989.

[12] D. A. Huffman, "A method for the construction of minimum-redundancy codes," *Proceedings of the IRE*, vol. 40, no. 9, pp. 1098–1101, 1952.

[13] Y. Li *et al.*, "Towards compact cnns via collaborative compression," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2021.

[14] Y. Zniyed, T. P. Nguyen *et al.*, "Hybrid network compression through tensor decompositions and pruning," in *2024 32nd European Signal Processing Conference (EUSIPCO)*. IEEE, 2024, pp. 1052–1056.

[15] T. Frederick and M. Greg, "Deep neural network compression by in-parallel pruning-quantization," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 6, 2018.

[16] X. Sun, E. Zhu, G. Qu, and F. Zhang, "Quadcnn: Quantized compression of deep cnn based on tensor-train decomposition with automatic rank determination," *Neurocomputing*, vol. 638, p. 130047, 2025.

[17] Y. Wen and D. Gregg, "Exploiting weight redundancy in cnns: Beyond pruning and quantization," *arXiv preprint arXiv:2006.11967*, 2020.

[18] G. I. Parisi, R. Kemker, J. L. Part, C. Kanan, and S. Wermter, "Continual lifelong learning with neural networks: A review," *Neural networks*, vol. 113, pp. 54–71, 2019.

[19] A. Krizhevsky, G. Hinton *et al.*, "Learning multiple layers of features from tiny images," 2009.

[20] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE conference on computer vision and pattern recognition*. IEEE, 2009, pp. 248–255.