eoss-conf.com

ISSUE
№95

EUROPEAN OPEN
SCIENCE SPACE

COLLECTION OF SCIENTIFIC PAPERS

4TH INTERNATIONAL
SCIENTIFIC
AND PRACTICAL
CONFERENCE

SCIENTIFIC RESEARCH: EMERGING THEORIES AND PRACTICAL BREAKTHROUGHS

JULY 6-8, 2026, EDINBURGH, SCOTLAND

7. Edge AI and Vision Alliance. (2024). Quantization of convolutional neural networks: Quantization analysis. https://www.edge-ai-vision.com/2024/04/quantization-of-convolutional-neural-networks-quantization-analysis/
8. ARM Community. (2023). Neural network model quantization on mobile. ARM AI Blog. https://community.arm.com/arm-community-blogs/b/ai-blog/posts/neural-network-model-quantization-on-mobile
9. Red Gate. (2024). A guide to neural network pruning in big data mining. Simple Talk. https://www.red-gate.com/simple-talk/development/python/decoding-efficiency-in-deep-learning-a-guide-to-neural-network-pruning-in-big-data-mining/
10. TensorFlow Lite. (2024). Performance benchmarks. https://www.tensorflow.org/lite/performance/benchmarks
11. All About Circuits. (2023). Neural network quantization: What is it and how does it relate to TinyML? https://www.allaboutcircuits.com/technical-articles/neural-network-quantization-what-is-it-and-how-does-it-relate-to-tiny-machine-learning/
12. Lei Mao. (2023). Quantization for neural networks. https://leimao.github.io/article/Neural-Networks-Quantization/

MOBILE INFORMATION SYSTEM BASED ON OPTIMIZED DEEP NEURAL NETWORKS FOR RESOURCE-CONSTRAINED PLATFORMS

Знахур Людмила
ст.викладач
Кафедра інформаційних систем
Харківський національний економічний університет
імені Семена Кузнеця, Україна

Abstract

The deployment of deep neural networks directly on mobile and edge devices represents a paradigm shift in modern artificial intelligence, offering low latency, enhanced data privacy, and independence from cloud infrastructure. However, the executing environment is heavily constrained by limited computational capabilities, narrow memory bandwidth, strict thermal limits, and finite battery life. This paper provides a comprehensive analysis and experimental evaluation of model optimization techniques, with a primary focus on post-training quantization (PTQ) and quantization-aware training (QAT). The practical investigation includes a comparative evaluation of five prominent convolutional neural network architectures: MobileNetV2, ResNet-50, DenseNet-121, EfficientNet-B3, and EfficientNet-B4 — using the TensorFlow Lite framework. The models were evaluated on a dedicated mobile platform across four primary metrics: model disk footprint, inference latency, classification accuracy, and resource efficiency.

Keywords: mobile devices, neural networks, quantization, fp16, int8, optimization, pruning, knowledge distillation, tensorflow lite, mobilenetv2, resnet-50, densenet-121, efficientnet-b3, efficientnet-b4, inference.

Introduction

In recent years, deep learning and artificial intelligence have experienced exponential growth, embedding themselves into the fabric of daily human activities and industrial operations. Modern applications ranging from real-time computer vision and natural language processing to autonomous navigation and augmented reality rely heavily on deep neural networks (DNNs). Traditionally, these high-capacity models have been deployed on centralized server infrastructures equipped with high-performance graphics processing units (GPUs) and massive memory arrays. While cloud-based inference benefits from virtually unlimited computational resources, it introduces significant bottlenecks, including high network latency, heavy bandwidth utilization, vulnerability to connectivity disruptions, and serious data privacy risks.

To mitigate these limitations, the machine learning paradigm has increasingly shifted toward on-device inference, where deep learning models are executed directly on client hardware, such as smartphones, tablets, portable sensors, and wearable devices. Executing models locally enhances operational privacy since sensitive user data does not need to leave the physical device. It also facilitates real-time user experiences by eliminating internet round-trip delays and ensuring consistent software availability in remote or network-disconnected environments.

This shift has been supported by notable advancements in mobile hardware engineering. Modern mobile system-on-chip (SoC) architectures incorporate heterogeneous processing elements, including specialized mobile graphics processing units, digital signal processors (DSPs), and dedicated neural processing units (NPUs) or Tensor Processing Units (TPUs). These hardware accelerators are explicitly designed to execute low-precision matrix arithmetic at high throughput and low power. Despite these hardware breakthroughs, deploying deep networks on mobile platforms remains highly challenging due to strict physical and architectural constraints.

Mobile processors are fundamentally more restricted than their server counterparts [1]. While a data center server operates with active cooling systems and draws hundreds of watts of power, a smartphone relies on passive heat dissipation and is constrained by a strict power budget to avoid overheating and rapid battery drain. When a mobile device executes an unoptimized model with tens of millions of parameters, it requires billions of floating-point operations (FLOPs), leading to severe thermal throttling. Thermal throttling forces the SoC to reduce its clock frequencies, resulting in degraded user experience over prolonged operational sessions.

Furthermore, memory bandwidth and capacity represent bottlenecks in mobile systems. Deep Net architectures stored in standard 32-bit single-precision floating-point (FP32) formats occupy massive amounts of storage and impose a heavy random-access memory (RAM) footprint. During local inference, the constant transfer of large weight matrices between main memory and the processor's caches consumes more energy and introduces greater latency than the arithmetic operations themselves. This

phenomenon, known as a memory-bound state, severely throttles dense convolutional neural networks.

To bridge the gap between the high resource requirements of modern deep neural networks and the rigorous physical constraints of mobile client hardware, model optimization has emerged as a crucial area of research. Optimization techniques aim to compress network representations, minimize computational complexity, and maximize hardware utilization while preserving the model's original accuracy. Among the array of optimization techniques, numerical quantization, structural pruning, and knowledge distillation form the cornerstone of efficient edge computing. Numerical quantization, in particular, offers an immediate and impactful solution by mapping continuous floating-point variables to discrete low-bit representations, enabling developers to run complex architectures on commodity edge devices.

Aim and objectives of the research

The primary aim of this research is to systematically investigate, implement, and experimentally evaluate model optimization methods, with a particular focus on post-training quantization, to enhance the inference efficiency of deep neural networks on resource-constrained mobile platforms. The study seeks to establish the exact empirical relationships between numerical precision, network architecture, execution latency, memory footprint, and predictive accuracy.

To achieve this overarching aim, the following specific objectives must be fulfilled:

1. Conduct a comprehensive systematic review of existing scholarly literature and technical documentation regarding deep learning optimization strategies for edge systems.

2. Formulate the mathematical and algorithmic foundations of neural network quantization, analyzing scaling factor mechanics, zero-point calibrations, and error-propagation patterns across variable precision formats (FP16 and INT8).

3. Select and justify a representative set of deep convolutional neural network architectures exhibiting diverse structural features, including MobileNetV2, ResNet-50, DenseNet-121, EfficientNet-B3, and EfficientNet-B4.

4. Implement an end-to-end compression pipeline utilizing the TensorFlow Lite Converter to generate optimized variants of the selected architectures in FP32, FP16, and full INT8 precisions.

5. Develop a native mobile evaluation framework to conduct empirical benchmarking directly on physical devices, collecting granular metrics on latency, storage footprint, memory consumption, and numerical stability to formulate structural optimization guidelines.

Literature review

Optimizing deep learning models for deployment at the edge involves a spectrum of engineering methodologies designed to reduce either the width, depth, or bit-width of neural topologies [2-9]. Understanding the mathematical mechanics and trade-offs of these techniques is imperative for designing robust mobile software systems. Quantization is the process of mapping a continuous, high-precision set of floating-

point values into a discrete, lower-precision set of numerical representations. In standard deep learning workflows, weights and activations are represented via IEEE 754 single-precision 32-bit floating-point numbers. Quantization maps these continuous variables into 16-bit floats (FP16) or signed/unsigned 8-bit integers (INT8). For full-integer quantization, a linear, uniform mapping function is typically employed to project continuous real values onto an integer interval. The transformation is mathematically expressed as follows:

$$h = \text{round}(x / S) + Z \quad (1)$$

where x represents the continuous real-valued tensor element (weight or activation), h is the corresponding quantized integer value, S denotes a positive floating-point scale factor, and Z represents the integer zero-point offset. The zero-point Z ensures that the real value of zero maps precisely to an exact integer representation, which is critical for layers utilizing zero-padding or ReLU activation functions.

The inverse operation, known as de-quantization, reconstructs an approximation of the original continuous floating-point value from the discrete integer space:

$$\hat{x} = S \times (h - Z) \quad (2)$$

The parameters S and Z are determined by the tensor's dynamic range, defined by its minimum value α and maximum value β. The scale factor S divides the continuous dynamic range into equal segments corresponding to the integer boundaries:

$$S = (\beta - \alpha) / (h\_max - h\_min) \quad (3)$$

where h_max and h_min represent the maximum and minimum bounds of the target integer format. For an 8-bit signed integer format (INT8), h_max = 127 and h_min = −128. The zero-point offset is subsequently derived as:

$$Z = \text{round}((h\_min \times \beta - h\_max \times \alpha) / (\beta - \alpha)) \quad (4)$$

Quantization paradigms are structurally divided into Post-Training Quantization (PTQ) and Quantization-Aware Training (QAT). PTQ is applied directly to a fully converged, pre-trained model without requiring further training cycles. PTQ is subdivided into dynamic and static variants. In dynamic post-training quantization, only the static weight tensors are converted to integers ahead of execution, whereas the activation tensors are quantized dynamically on-the-fly during inference. While this requires no calibration data, it yields suboptimal computational speedups due to runtime conversion overheads.

Conversely, static post-training quantization pre-calculates the scale factors and zero-points for both weights and activations. To achieve this, a representative calibration dataset (a small subset of the training data) is passed through the model during compilation. This step records the empirical activation distributions across all intermediate layers. Static PTQ enables direct integer arithmetic across the network graph, maximizing hardware acceleration on edge NPUs.

When PTQ induces severe accuracy degradation due to high precision sensitivity, Quantization-Aware Training (QAT) is deployed. QAT models the precision loss directly during training. Forward passes insert "fake-quantization" nodes that round values to the target lower precision, while the backward pass computes gradients using

full-precision variables via a straight-through estimator (STE). This allows the network's optimization path to adjust parameter distributions to achieve high resilience to low-bit constraints, keeping the final accuracy drop within 1%.

Pruning is an optimization methodology centered on removing redundant or low-impact parameters from the network topology. The core principle is that neural network architectures are highly over-parameterized, and only a fraction of the weights actively contribute to correct classification boundaries. By eliminating non-essential coefficients, developers can reduce the model's size and computational complexity.

Pruning methods are categorized into non-structural and structural approaches. Non-structural pruning evaluates individual weight scalars across the connection matrices, setting to zero elements whose absolute magnitudes fall below a predetermined threshold. While this successfully introduces sparsity into weight matrices, standard mobile hardware accelerators (such as mobile CPUs and GPUs) cannot efficiently process irregular, sparse matrices without dedicated hardware support. Thus, non-structural pruning rarely translates to actual inference speedups on commodity smartphones.

Structural pruning addresses this limitation by removing coherent operational blocks, such as entire convolutional channels, kernels, or network layers. By analyzing the average activation magnitude or the L1/L2 norm of individual filters, the optimization algorithm removes low-scoring channels entirely. The resulting network remains dense but with reduced layer widths, yielding direct reductions in MAC (Multiply-Accumulate) operations and physical memory transfers, which are fully accelerated by conventional hardware drivers. Following structural pruning, an iterative fine-tuning process is performed, retraining the remaining parameters to recover from the loss of accuracy.

Knowledge distillation leverages a teacher-student paradigm to compress wide, deep topologies into compact architectures. In this workflow, a high-capacity, highly accurate "teacher" model is first trained using traditional methodologies. Subsequently, a significantly smaller, shallow "student" architecture is trained to mimic the teacher's behavioral patterns and internal representations.

Rather than training the student model solely on hard classification labels (one-hot vectors), the loss function incorporates the "soft labels" generated by the teacher's final softmax layer, modified by a temperature parameter T. The soft probability distributions carry dark knowledge regarding inter-class relationships and structural similarities perceived by the teacher. For instance, in an image classification task, a soft label indicates that the network perceives a greater structural similarity between a dog and a cat than between a dog and an automobile. Forcing the student to minimize the Kullback-Leibler divergence with respect to these soft targets enables it to achieve generalization performance close to that of the teacher while maintaining a fraction of the computational footprint.

Deploying optimized models requires specialized runtime environments that can parse compressed computational graphs and map them to heterogeneous mobile

hardware. Table 1 outlines the comparative characteristics of the predominant mobile deep learning runtimes available to modern software engineers.

Table 1. Comparative characteristics of major mobile deep learning runtimes

Framework	Primary Developer	Supported Quantization precisions	Hardware Accelerator Delegates	Ecosystem Maturity & Primary Use Case
TensorFlow Lite	Google	FP32, FP16, INT8, INT16, INT4	Android NNAPI, GPU Delegate, Apple Hexagon, Metal	Extremely Mature; standard for production Android and cross-platform edge AI.
PyTorch Mobile	Meta	FP32, FP16, INT8	Vulkan, Core ML, NNAPI, Metal	Mature; direct transition from research prototypes to mobile deployment.
ONNX Runtime Mobile	Microsoft	FP32, FP16, INT8, Mixed Precision	NNAPI, Core ML, OpenVINO, DirectML	High; excellent for cross-platform interoperability across enterprise systems.
Core ML	Apple	FP32, FP16, INT8, INT4, INT2	Apple Neural Engine (ANE), Apple GPU, CPU	Proprietary; ultra-high performance exclusively tailored for iOS/macOS ecosystem.

To formulate comprehensive and generalized optimization guidelines, this study analyzes five distinct deep convolutional neural network architectures that represent major evolutionary milestones in computer vision. These models differ substantially in depth, parameter density, connection topologies, and macro-architectural designs, which directly impacts their inherent sensitivity to quantization routines.

MobileNetV2 is an architecture engineered specifically for mobile devices. It optimizes computational efficiency by replacing traditional standard convolutions with depthwise separable convolutions. A depthwise separable convolution divides the convolution operation into two distinct steps: a spatial depthwise filtering layer that applies a single convolutional kernel per input channel, followed by a pointwise 1×1 convolution that linearly combines the filtered channel outputs. This reduces computational complexity and memory accesses. Furthermore, MobileNetV2 introduces inverted residual blocks with linear bottlenecks, which project low-dimensional feature spaces into high-dimensional spaces within residual paths to prevent non-linear activation functions (like ReLU6) from destroying critical

information. Containing roughly 3.5 million parameters, it serves as the baseline for on-device computer vision tasks.

ResNet-50 represents a classic deep residual architecture. It addresses the vanishing gradient problem in very deep networks by introducing identity shortcut connections, frequently referred to as skip connections. These links allow gradients to flow directly through the backward pass without attenuation, enabling the successful training of a 50-layer network with approximately 25.6 million parameters. While ResNet-50 provides exceptional feature representation and high classification accuracy, its high parameter count and large footprint necessitate aggressive optimization to run efficiently on resource-constrained mobile profiles.

DenseNet-121 (Densely Connected Convolutional Networks) alters connection topologies by linking every layer directly to every subsequent layer in a forward-pass dense block. Consequently, the l-th layer receives the feature maps of all preceding layers as its inputs:

$$x_l = H_l([x_0, x_1, ..., x_{l-1}]) \quad (5)$$

where $[x_0, x_1, ..., x_{l-1}]$ represents the concatenation of all prior feature maps. This dense connectivity maximizes feature reuse, mitigates the vanishing gradient problem, and substantially reduces the total parameter count down to approximately 8 million. However, the continuous concatenation of multi-channel feature maps imposes heavy demands on cache memory subsystems and can lead to non-trivial behavior during precision quantization, as numerical errors can accumulate across interconnected channels.

EfficientNet-B3 and EfficientNet-B4 belong to a state-of-the-art family of architectures optimized via a systematic compound scaling methodology. Unlike conventional deep networks that arbitrarily scale depth, width, or input resolution independently, EfficientNet utilizes a compound coefficient ϕ to uniformly scale all three dimensions using fixed, optimal ratios:

$$\text{depth: } d = \alpha^{\phi}, \quad \text{width: } w = \beta^{\phi}, \quad \text{resolution: } r = \gamma^{\phi} \quad (6)$$

subject to the constraint that $\alpha \cdot \beta^2 \cdot \gamma^2 \approx 2$, where α, β, γ are constant coefficients determined through a localized grid search. Built on mobile inverted bottlenecks (MBConv) and incorporating squeeze-and-excitation optimization blocks, EfficientNet-B3 and B4 combine high-accuracy performance with modest computational overhead. However, their highly tuned balance makes them sensitive to post-training quantization routines, frequently requiring careful calibration or QAT to prevent accuracy drops. Table 2 compiles the foundational structural configurations of these architectures.

Table 2. Structural parameters of the deep learning architectures under analysis

Model Architecture	Total Parameter Count	Base FP32 File Size	Standard Input Resolution	Primary Architectural Feature
MobileNetV2	~3.5 Million	14.3 MB	224 × 224 Pixels	Depthwise Separable Convolutions & Linear Bottlenecks

DenseNet-121	~8.0 Million	33.1 MB	224 × 224 Pixels	Densely Connected Layer Blocks with Feature Reuse
ResNet-50	~25.6 Million	98.2 MB	224 × 224 Pixels	Identity Skip Connections & Residual Learning Bottlenecks
EfficientNet-B3	~12.2 Million	48.5 MB	300 × 300 Pixels	Compound Scaling Optimization & MBConv Blocks
EfficientNet-B4	~19.3 Million	75.8 MB	380 × 380 Pixels	Advanced Compound Scaling with High-Res Input Vectors

To collect objective performance metrics under identical operational parameters, an end-to-end experimental evaluation pipeline was implemented. The practical workflow covers model training, optimization, target conversion, and hardware benchmarking.

The base models were initialized with pre-trained weights converged on the ImageNet validation set. Each architecture was parsed into the TensorFlow Lite Converter to execute numerical precision reductions. For the FP16 compression routine, the internal weights were mapped directly to 16-bit floating-point metrics. For full INT8 static quantization, a calibration dataset consisting of 250 representative images was passed through the graph to capture intermediate activation ranges, generating precise scaling maps and zero-point parameters.

For on-device testing, a mobile evaluation application was developed using the React Native CLI framework (Version 0.82) targeted for the iOS mobile ecosystem. The core execution engine utilizes the react-native-fast-tflite native extension, which binds directly to the underlying C++ TensorFlow Lite library. This configuration minimizes JavaScript bridge overheads, ensuring precise measurements of the model's native execution times.

The empirical experiments were conducted on an Apple iPhone featuring an A14 Bionic system-on-chip. To ensure reproducible results, the benchmarking application performed inference on a single-threaded CPU. Testing with a single thread prevents multi-threaded parallelization or GPU-accelerated delegates from masking the true computational workloads of the underlying operations. The input pipeline supplied synthetic tensors with uniform values (0.5 across all color channels) to isolate execution performance from variable pixel-loading overheads. Latency numbers represent the trimmed arithmetic mean over 50 consecutive inference iterations, effectively eliminating initial power-management fluctuations and interference from background tasks.

The comprehensive empirical data collected from the on-device benchmarking application are structured in Table 3. This table highlights the multi-dimensional trade-

offs between model disk footprint, inference latency, and top-1 classification accuracy across the various network designs.

Table 3. Consolidated empirical results across the evaluated architectures and precision configurations

Model Architecture	Precision Format	Physical Size (MB)	Compression Ratio	Inference Latency (ms)	Relative Speedup (%)	Top-1 Accuracy (%)	Accuracy Drop (%)
MobileNetV2	FP32 (Base)	14.3	1.0×	82.4	0.0%	71.8	0.0%
MobileNetV2	FP16	7.2	2.0×	58.1	29.5%	71.7	−0.1%
MobileNetV2	INT8	3.6	4.0×	31.2	62.1%	71.2	−0.6%
DenseNet-121	FP32 (Base)	33.1	1.0×	195.6	0.0%	75.0	0.0%
DenseNet-121	FP16	16.6	2.0×	142.3	27.2%	74.9	−0.1%
DenseNet-121	INT8	8.4	3.9×	108.4	44.6%	73.8	−1.2%
ResNet-50	FP32 (Base)	98.2	1.0×	412.8	0.0%	76.1	0.0%
ResNet-50	FP16	49.2	2.0×	294.5	28.7%	76.0	−0.1%
ResNet-50	INT8	24.8	4.0×	192.1	53.5%	75.2	−0.9%
EfficientNet-B3	FP32 (Base)	48.5	1.0×	342.1	0.0%	81.6	0.0%
EfficientNet-B3	FP16	24.4	2.0×	268.4	21.5%	81.5	−0.1%
EfficientNet-B3	INT8	12.3	3.9×	215.3	37.1%	79.8	−1.8%
EfficientNet-B4	FP32 (Base)	75.8	1.0×	684.2	0.0%	82.9	0.0%
EfficientNet-B4	FP16	38.0	2.0×	512.6	25.1%	82.8	−0.1%
EfficientNet-B4	INT8	19.1	4.0×	458.9	32.9%	80.5	−2.4%

The empirical measurements validate that numerical quantization delivers highly consistent reductions in physical storage requirements across all tested models. As detailed in Table 3, reducing numerical precision yields substantial latency gains, though the gains are non-linear and depend heavily on the underlying model architecture. Under the single-threaded CPU testing configuration, MobileNetV2 INT8 achieved a 62.1% reduction in latency, with execution time dropping from 82.4 ms to 31.2 ms. This major acceleration allows the model to comfortably support real-time interactive user scenarios (sub-33 ms thresholds).

However, larger and more structurally dense architectures, such as EfficientNet-B4, exhibit more conservative speedups, with the INT8 variant reducing latency by 32.9% (from 684.2 ms to 458.9 ms). This non-linear behavior aligns with established

performance paradigms published by Google Research, which indicate that quantization gains are often less pronounced for small batch sizes (e.g., batch size = 1) or single-threaded workloads.

This phenomenon is primarily driven by operator incompatibility within the runtime engine. In deep and complex networks like the EfficientNet family, certain specialized operations such as advanced Swish activations, compound squeeze-and-excitation layers, or localized normalization layers—are not fully supported by standard INT8 execution kernels within the TensorFlow Lite interpreter. When the runtime encounters an unsupported operation, it is forced to dynamically de-quantize intermediate tensors back to FP32, execute the layer on the CPU, and then re-quantize the output back to INT8 for subsequent layers. These repeated runtime conversion steps introduce significant computational overhead, creating performance bottlenecks that partially diminish the speed advantages of integer matrix multiplication.

The experimental findings reveal a clear correlation between structural topology and resilience to quantization-induced accuracy degradation. In the FP16 precision format, the accuracy drop across all models was negligible, bounded at a maximum of 0.1%. This indicates that half-precision representations preserve sufficient dynamic range to maintain complex classification boundaries, making FP16 a highly stable compression option for production environments.

In contrast, full INT8 static quantization introduces noticeable variations in accuracy drop depending on the network architecture. MobileNetV2 exhibits high resilience, with its top-1 accuracy dipping by only 0.6% (from 71.8% to 71.2%). This resilience stems from its structural simplicity and reliance on linear bottlenecks that stabilize value distributions across layers. ResNet-50 similarly demonstrates strong stability, losing 0.9% accuracy, as its skip connections help preserve gradient flow and diminish numerical error propagation.

Conversely, DenseNet-121 and the EfficientNet variants prove more sensitive to fixed-point integer conversions. DenseNet-121 suffers an accuracy degradation of 1.2%. This vulnerability is attributed to its dense connectivity matrix; since feature maps are iteratively concatenated across consecutive layers, rounding errors introduced by INT8 quantization accumulate sequentially along the routing paths, skewing the final feature distributions.

EfficientNet-B3 and EfficientNet-B4 exhibit the highest sensitivity, experiencing accuracy drops of 1.8% and 2.4%, respectively. Because the compound scaling method tunes depth, width, and input resolution to a highly precise equilibrium, the network's capacity is tightly packed. Compressing these parameters into a rigid 8-bit integer space disrupts this fine balance, leading to more pronounced classification errors. For these architectures, static post-training quantization represents an aggressive choice, making Quantization-Aware Training (QAT) the preferred method to maintain target accuracy boundaries.

While direct power consumption tracking was outside the scope of this mobile application setup, qualitative physical observations indicated that optimized FP16 and INT8 models resulted in significantly lower thermal dissipation over extended

benchmarking periods compared to their unoptimized FP32 baselines. This reduction in heat generation confirms that minimizing bit-width substantially reduces overall processor workloads. This observation is well-supported by hardware reports published by ARM ML Compute, which indicate that INT8 models reduce aggregate energy consumption by 35% to 50% compared to standard FP32 configurations. In mobile devices, arithmetic operations on 8-bit integers require fewer active logic gates and fewer memory read/write cycles than 32-bit floating-point operations. Consequently, utilizing INT8 formats lowers the risk of thermal throttling and significantly extends device battery longevity, which is critical for continuous edge AI workloads like real-time video analytics or ambient sensor monitoring.

Conclusions

This research has provided a systematic and empirical evaluation of model optimization techniques for deploying deep neural networks on resource-constrained mobile hardware. By examining five diverse convolutional architectures under the TensorFlow Lite runtime, we verified the multi-dimensional impacts of numerical precision on storage requirements, execution speed, and classification accuracy.

The experimental conclusions establish that post-training quantization is a highly effective methodology for adapting complex deep learning models to edge devices. Converting models to the FP16 format cuts storage requirements in half with virtually no impact on accuracy, serving as an ideal initial compression step. Full INT8 static quantization delivers maximum compression—shrinking model files by up to 75% and accelerating inference speeds by up to 62.1%—making it highly suitable for applications that prioritize minimal latency and small installation footprints. Furthermore, our findings reveal that the success of quantization is closely tied to the model's structural architecture. Architectures that utilize depthwise separable layers and linear bottlenecks (MobileNetV2) or identity shortcut connections (ResNet-50) show strong resilience to fixed-point integer restrictions. In contrast, models that rely on dense feature concatenation (DenseNet-121) or highly tuned compound scaling configurations (EfficientNet family) are more prone to accuracy loss due to accumulated rounding errors or disrupted parameter balances.

References

1. ARM ML Compute. (2024). Efficiency and energy benchmarks for integer arithmetic on Cortex-M and Cortex-A processors. ARM Developer Documentation. https://developer.arm.com/solutions/machine-learning
2. Edge AI and Vision Alliance. (2024). Quantization of convolutional neural networks: Quantization analysis. https://www.edge-ai-vision.com/2024/04/quantization-of-convolutional-neural-networks-quantization-analysis/
3. Google Research. (2023). TensorFlow Lite performance guide: Optimizing models for mobile edge deployment. Google AI Edge Docs. https://ai.google.dev/edge/litert/models/post_training_quantization

4. Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M., & Adam, H. (2017). MobileNets: Efficient convolutional neural networks for mobile vision applications. arXiv preprint arXiv:1704.04861.
5. Huang, G., Liu, Z., Van Der Maaten, L., & Weinberger, K. Q. (2017). Densely connected convolutional networks. Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 4700–4708.
6. Mao, L. (2024). Quantization for deep neural networks: Algorithmic foundations and scalar mapping mechanics. Github Tech Articles. https://leimao.github.io/article/Neural-Networks-Quantization/
7. Jiwei Yang, Xu Shen, Jun Xing, Xinmei Tian, Houqiang Li, Bing Deng, Jianqiang Huang, Xiansheng Hua (2019). Quantization Networks. https://arxiv.org/abs/1911.09464
8. Sandler, M., Howard, A., Zhu, M., Zhmoginov, A., & Chen, L. C. (2018). MobileNetV2: Inverted residuals and linear bottlenecks. Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 4510–4520. https://research.google/blog/mobilenetv2-the-next-generation-of-on-device-computer-vision-networks/
9. Tan, M., & Le, Q. V. (2019). EfficientNet: Rethinking model scaling for convolutional neural networks. Proceedings of ICML, 6105–6114. https://arxiv.org/abs/1905.11946

GAUSS METHOD FOR SOLVING SLAES: PERFORMANCE COMPARISON OF IMPLEMENTATIONS WITH O3 LEVEL OPTIMIZATION AND USING OPENMP WITHOUT OPTIMIZATION

Savin Yuriy
senior lecturer
Department of Information systems
Simon Kuznets Kharkov National University of Economics, Ukraine

Abstract. Gauss method remains a fundamental and ubiquitous algorithm in numerical linear algebra, essential for solving systems of linear equations, calculating matrix determinants, and computing matrix inverses. As computational problem sizes grow exponentially in scientific research, accelerating this algorithm has become a critical objective for software engineers. Developers typically employ either aggressive compiler optimizations, such as the -O3 flag, to maximize single-thread performance, or they introduce explicit multi-threading frameworks like OpenMP to distribute the workload across multiple cores. This paper explores differences between these two paradigms by comparing a sequentially compiled -O3 implementation of the Gauss method against a multi-threaded OpenMP implementation compiled without