Sarcouncil Journal of Engineering and Computer Sciences

ISSN(Online): 2945-3585

Volume- 04| Issue- 11| 2025

Research Article **Received:** 20-10-2025| **Accepted:** 15-11-2025 | **Published:** 28-11-2025

Post-Training Optimization Techniques for AI Models: A Comprehensive Framework

Reeshav Kumar

Independent Researcher, USA

Abstract: Post-training optimization techniques play a crucial role in transforming trained AI models into practical systems that can be deployed in production. The article introduces a strata model that combines model, runtime, and system-level strategies to assist AI practitioners in developing high-performance systems that can efficiently address resource constraints. Post-training quantization (PTQ), sparsity pruning, low-rank adaptation (LoRA), and knowledge distillation are some of the model-level methods that help improve parameter efficiency. Compiler-time optimizations include fusion of operators, restructuring of memory layout, constant folding, auto-tuning of kernels, and compiler re-architecture to improve computational efficiency. System-level strategies, such as dynamic batching, KV cache reuse, paged attention, request coalescing, and model routing, are used to optimize models for a particular deployment environment, enabling efficient resource utilization and an optimal user experience. The article introduces a systematic approach to trade-off analysis between quality and latency, throughput and memory, energy consumption, and cost, which enables AI practitioners to make informed decisions based on optimization, ensuring efficiency and reliability in various applications and serving infrastructure.

Keywords: Post-Training Optimization, Model Quantization, Parameter Efficiency, Deployment Frameworks, Inference Acceleration.

INTRODUCTION

Implementing AI models in a production setting is a significant challenge facing other organizations in various fields that aim to adopt AI in their operations. Although most attention is usually paid to the model architecture and training protocols, post-training optimization determines the model's success in real-life applications. Post-training optimization methods enhance deployment capabilities, reduce operational costs, and improve experience without requiring complete retraining of the model. This paper provides a structured overview of post-training optimization methods over three critical layers of the AI deployment stack. Each layer addresses separate topics in model optimization, providing practitioners with a comprehensive arsenal for transforming research prototypes into deployable systems.

As models grow considerably in size and complexity, the gap between research prototypes and deployable systems increases, an effect often referred to as the “deployment efficiency barrier”. For example, in healthcare applications, optimizations are driven by strict computational capacity and latency requirements for clinical decision-support systems. In financial services, the emphasis is typically on throughput optimization to handle high transaction volumes while maintaining strict regulatory compliance (Ahmed, A. *et al.*, 2025).

Recent studies have identified significant variation in the effectiveness of optimization techniques for hardware targets and model structures. The proposed three-tier framework, comprising model-level, runtime-level, and system-level techniques, addresses the complex multidimensionality of deployment challenges, encompassing parameter effectiveness, computational efficiency, and system-level resource allocation. Organizations that deploy a formalized approach across the software and hardware stack see more predictable results than those using optimization methods in standalone form or without strategic prioritization throughout the deployment stack (Ahmed, A. *et al.*, 2025).

The larger industry ecosystem has also come to accommodate these optimization workflows, with each layer of the optimization framework having its own dedicated tooling. The advent of specialized optimization platforms is a response to the increasing awareness that deployment efficiency is a key competitive differentiator in aggressive AI markets. These toolchains more often include automated benchmarking systems that assess various optimization permutations against application-specific performance requirements. The corresponding empirical performance profiles facilitate better-informed decision-making about optimization trade-offs, particularly in sophisticated multi-stakeholder contexts where varying performance

***Corresponding Author:** Reeshav Kumar

DOI- https://doi.org/10.5281/zenodo.17754346

characteristics can be prioritized by different organizational functions (Sinha, S., & Lee, Y. M. 2024).

Aside from short-term performance gains, overall optimization practices yield substantial operational benefits. Organizations that use systematic post-training optimization experience a speedup in deployment cycles, with considerably lower average duration from model validation to production deployment. This speedup is not just a function of more efficient models but also of deploying process standardization and improved production predictability. Moreover, improvements in resource efficiency translate directly into sustainability gains, as optimized models exhibit significantly lower energy consumption profiles throughout their working lifetimes, a growing priority for carbon-conscious organizations (Sinha, S., & Lee, Y. M. 2024).

Fig. 1. Competing dimensions for post-training optimization

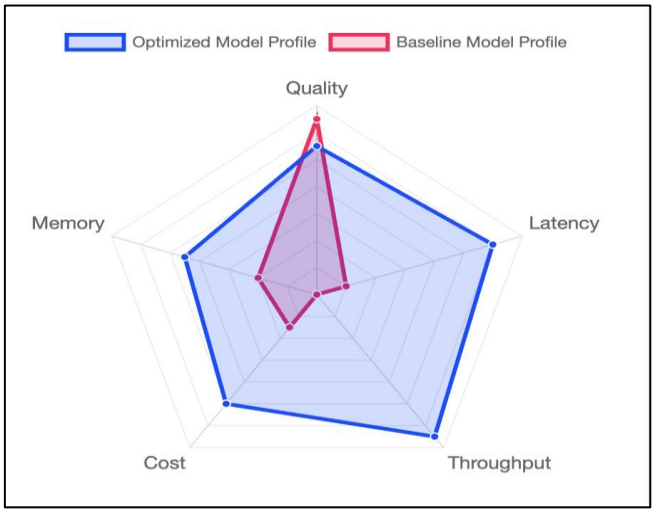

Fig 2. Post-training optimization balances competing objectives to produce scalable and reliable models suitable for large-scale deployment.

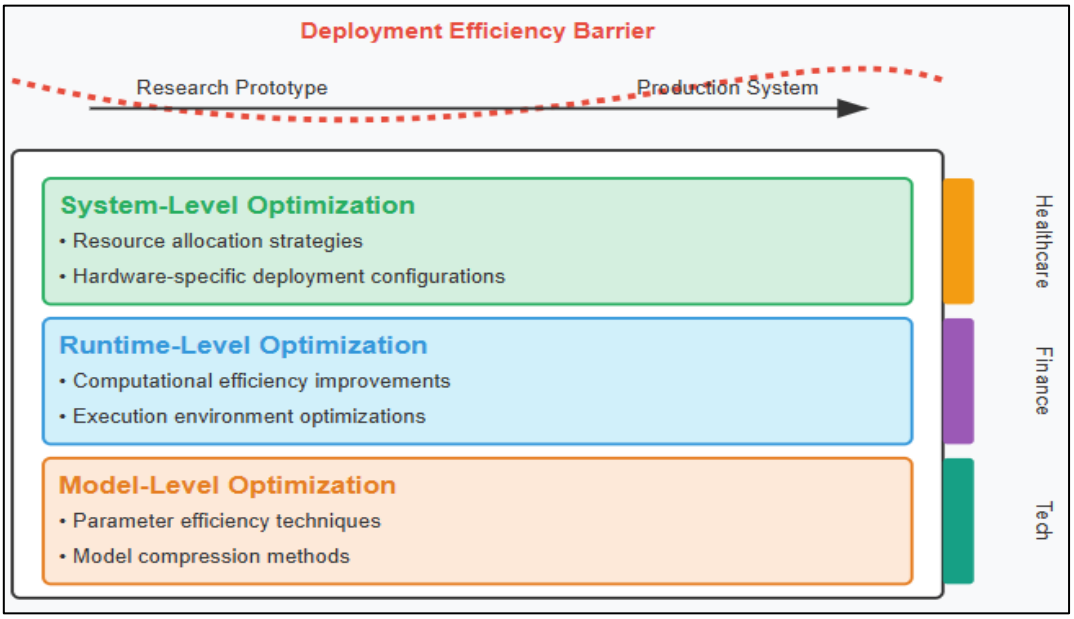

Fig 3: Post-Training Optimization Framework (Ahmed, A. *et al.*, 2025; Sinha, S., & Lee, Y. M. 2024)

Publisher: SARC Publisher

MODEL-LEVEL OPTIMIZATION TECHNIQUES

Model-level optimizations are used to modify the architecture of the trained model and optimize its parameters, thereby enhancing performance without altering its functional behavior. These methods act immediately on the model's inner representation.

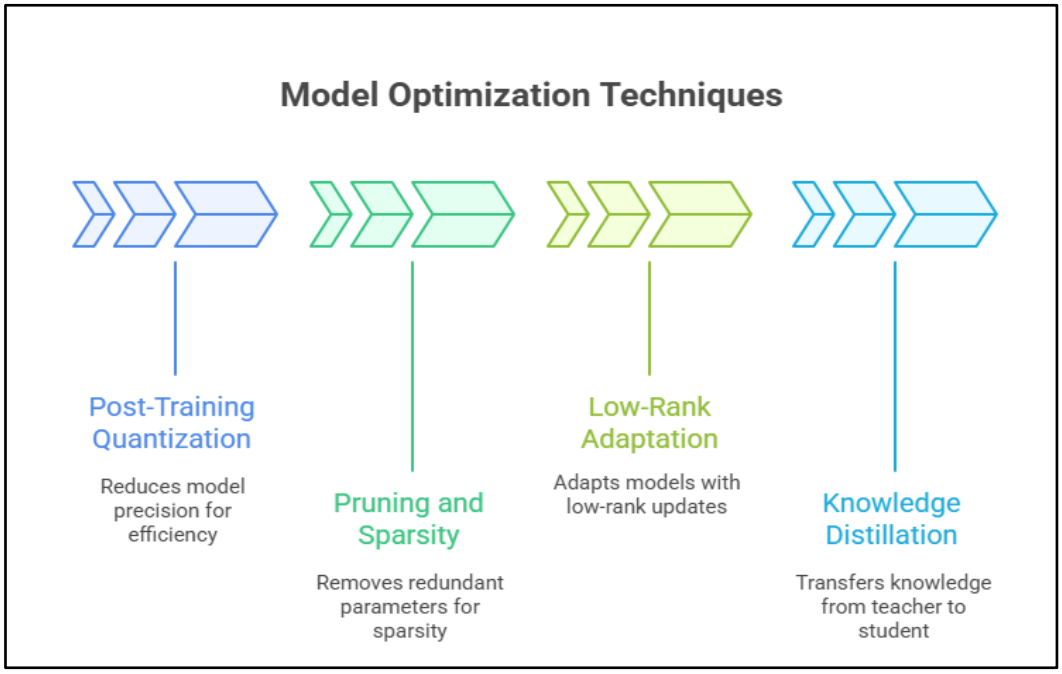

Fig 4: Model Optimization Techniques (Liang, T. *et al.*, 2021; Houlsby, N. *et al.*, 2019)

Post-Training Quantization

Post-training quantization (PTQ) reduces the numerical accuracy of model weights and activations from higher-precision data types (typically 32-bit floating-point) to lower-precision data types (8-bit integers, 16-bit floating-point, or even 4-bit or binary) without compromising inference accuracy to unacceptable levels.

Recent breakthroughs in post-training quantization have demonstrated significant efficiency improvements across a wide range of model architectures. Quantization methods have been especially beneficial for transformer models, which tend to be memory bandwidth and computationally limited. Examination of different quantization granularities reveals that per-channel methods tend to yield the best trade-off between quantization effectiveness and accuracy retention, particularly for intricate networks with strongly differing activation distributions across channels. Empirical tests demonstrate that the calibration strategy has a strong effect on quantization results. Methods that record the entire distribution of activations for representative inference passes outperform simpler approaches based on tensor extrema. A quantization-aware fine-tuning framework, which accounts for the effects of quantization during constrained retraining periods, is beneficial for restoring accuracy in sensitive parts of the network, such as attention mechanisms and early convolutional layers, where quantization information loss can propagate throughout the network (Liang, T. *et al.*, 2021).

The workaday deployment advantages of quantization go beyond theoretical reductions in compute. New hardware accelerators are also being optimized for low-precision arithmetic, with custom matrix multiplication units for INT8 and mixed-precision computation. This hardware-software co-optimization enables achieved throughput gains that closely approach theoretical expectations, rather than other optimization solutions where theoretical and actual gains widely diverge. Energy efficiency gains due to quantization are equally dramatic, with INT8 computing generally consuming 4-16x less energy than FP32 computing on particular hardware architectures. These efficiency benefits have been significant for edge deployment use cases with hard power limits, supporting on-device inference of models that would otherwise necessitate cloud connectivity, along with attendant latency and privacy concerns (Liang, T. *et al.*, 2021).

Publisher: SARC Publisher

Pruning and Sparsity

Pruning methods remove redundant or unimportant parameters from neural networks, resulting in sparse models that consume fewer computational resources and memory. When coupled with sparsity-enabled kernels, pruned models can realize significant acceleration.

The neural network pruning phenomenon is related to several fundamental questions regarding deep network overparameterization and representation learning. The "lottery ticket hypothesis" has helped provide a theoretical context, implying that sparse subnetworks within dense neural networks can achieve comparable performance upon training alone, provided they are initialized from the right points. This thinking has influenced practical pruning methods, particularly iterative magnitude pruning techniques that alternate between training and parameter elimination steps. Structured pruning methods have gained popularity in production environments due to their ability to integrate with existing acceleration libraries, although they typically yield lower theoretical compression ratios than unstructured methods. Channel pruning from convolutional networks and attention head pruning from transformers are potent structured methods, eliminating entire computational units instead of individual weights, and thus allowing for acceleration directly without requiring expertise in sparse computation libraries (Houlsby, N. *et al.*, 2019).

The interplay of pruning schedules, learning rate behavior, and generalization characteristics has been the focus of growing interest. The evidence suggests that gradual pruning methods, which eliminate parameters across multiple iterations, tend to maintain accuracy better than one-shot pruning strategies, especially at higher sparsity levels. Pruning performance also varies significantly across different network topologies and layers, with shallower network layers generally being more susceptible to pruning than deeper layers. Non-uniform pruning policies are therefore utilized to maximize various sparsity goals across different network elements, along with sensitivity analysis. Pruning, combined with other forms of optimization, must be carefully coordinated—for example, whether quantization and pruning work together or against each other depends on the actual implementation strategy and the order of application (Houlsby, N. *et al.*, 2019)].

Low-Rank Adaptation

Low-rank adaptation (LoRA) methods enable effective domain adaptation without requiring complete model retraining. By breaking down weight updates into low-rank approximations, such methods provide parameter-efficient fine-tuning that accounts for domain shifts while preserving the essential abilities of the pre-trained model.

The advent of low-rank adaptation approaches is a paradigm shift in adapting models for individual domains or tasks. LoRA relies on basic linear algebraic intuition, using the fact that fine-tuning updates tend to fall in a low-dimensional subspace of the entire parameter space. By limiting updates to this subspace through low-rank decomposition, such approaches significantly decrease the number of trainable parameters while maintaining the capacity for adaptation. The method introduces two tiny matrices per adapting weight matrix: a down-projection matrix and an up-projection matrix, whose product is a low-rank update to the native weights. The rank hyperparameter offers a straightforward control knob for the adaptation-efficiency tradeoff, with empirical evidence indicating that remarkably low ranks (usually 4-32) approximately capture most task-specific adaptations for even large models with billions of parameters (Liang, T. *et al.*, 2021).

The practical benefits of LoRA deployment extend beyond computational performance when fine-tuning. The method enables a modular adaptation architecture, where a single base model can be paired with various lightweight adaptation modules for multiple domains or tasks. Such modularity greatly minimizes storage demands and eases deployment architectures, as the base model parameters remain constant across all adaptations. The low-rank form also yields built-in regularization, which can enhance domain-specific generalization compared to full fine-tuning, especially in low-data settings. Combining these methods with other approaches, such as quantization and pruning, also requires consideration of interaction effects. Although recent work suggests that these methods complement each other when sequenced appropriately (Liang, T. *et al.*, 2021), further investigation is needed.

Knowledge Distillation

Knowledge distillation transfers learning from a large, complex model (the teacher) to a smaller, more efficient model (the student). It utilizes the soft probability distributions generated by the

Publisher: SARC Publisher

teacher to inform the training of the student, resulting in compact models that perform similarly to much larger counterparts.

Knowledge distillation addresses the issue of model capacity compression while maintaining performance, leveraging the insight that teacher output distributions contain more information than mere hard labels. The soft probabilities assigned to non-target classes convey useful similarity information, directing student training towards more nuanced decision boundaries. Temperature scaling of the softmax function is a crucial control mechanism in distillation, where temperatures yield smoother probability distributions that emphasize the relative relationships among classes. In addition to classification logits, modern distillation methods employ multiple mechanisms for knowledge transfer, including intermediate feature representations and attention maps. Feature distillation is especially useful in deeper networks, where teacher internal representations learn hierarchical abstractions that are useful for student learning (Houlsby, N. *et al.,* 2019)].

The actual application of distillation in production environments involves several critical design choices that significantly impact efficacy. The structural relationship between teacher and student has a crucial effect on the outcome of distillation, with students who retain comparable structural patterns to those of their teachers but decrease the width or depth generally performing more effectively at transferring knowledge than those with entirely distinct architectures. Distillation goals typically balance a supervised loss on difficult labels and a knowledge transfer loss, which aims to prevent divergence between the teacher and learner distributions. The trade-off between these two is a vital hyperparameter whose optimal values depend on task difficulty and data properties. Online distillation methods, where students continue to learn in parallel with the teacher, have shown promising outcomes for specific applications, albeit at the expense of more complex training. Placing distillation within larger optimization pipelines typically leaves it as a terminal step, after architecture choice and early training. However, recent studies indicate that incorporating distillation guidance throughout the entire training process has advantages (Houlsby, N. *et al.*, 2019)

Table 1: Efficiency vs. Accuracy Trade-offs in Model-Level Optimizations (Liang, T. *et al.,* 2021; Houlsby, N. *et al.,* 2019)

Technique	Primary Benefit	Memory/Compute Impact	Performance Trade-off	Hardware Dependency
Post-Training Quantization	Reduced precision (FP32→INT8/FP16)	4-16× less energy consumption	Minimal accuracy impact with calibration	High (hardware acceleration)
Pruning & Sparsity	Parameter reduction	Smaller model footprint	Structured pruning: less compression but direct acceleration	Medium (sparsity-aware kernels)
Low-Rank Adaptation (LoRA)	Parameter-efficient fine-tuning	Only 0.1-1% parameters updated	Rank hyperparameter controls adaptation-efficiency trade-off	Low
Knowledge Distillation	Model size reduction	Smaller student model	Temperature scaling affects knowledge transfer	Low

Runtime-Level Optimization Methods

Runtime optimizations aim to increase computational efficiency in model execution through several transformations and optimizations.

Publisher: SARC Publisher

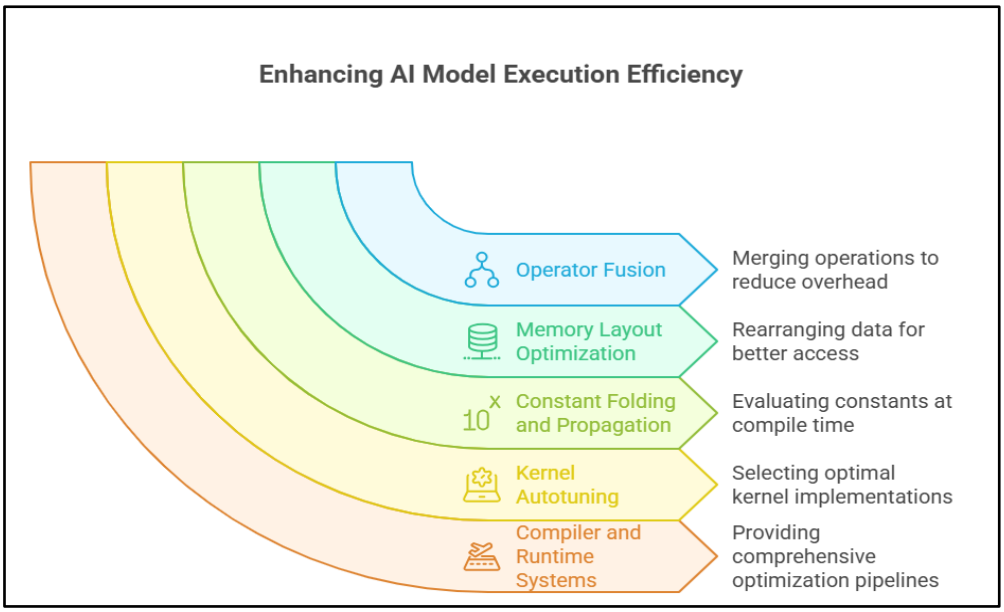

Fig 5: Enhancing AI Model Execution Efficiency (Chen, T. *et al.*, 2018; Lattner, C. *et al.*, 2021)

Operator Fusion

Operator fusion merges consecutive operations into a single, optimized kernel to minimize memory transfers and kernel launch overhead. The method is efficient in cases where operations are frequently chained together, such as convolution followed by batch normalization and activation functions.

The performance improvement from operator fusion primarily comes in the form of reduced memory traffic and kernel launch overhead. Most deep learning frameworks construct individual layers as individual operators and thus incur a redundant round-trip to global memory for intermediate computations. The XLA compiler addresses this inefficiency with vertical fusion, which merges sequential operations in the computation graph, and horizontal fusion, which parallelizes independent operations to utilize hardware more effectively. Analysis across typical neural network structures demonstrates that fusion can reduce memory bandwidth requirements by 3 times and noticeably decrease kernel launch overhead, which can otherwise overwhelm execution time for minor operations. Instrumental fusion scenarios are element-wise operations after matrix multiplications or convolutions, where fusion avoids materialization of huge intermediate tensors. The complexity of implementing fusion depends dramatically on the hardware target, with GPUs typically requiring more complex memory handling to achieve optimal occupancy and avoid register spilling (Chen, T. *et al.*, 2018).

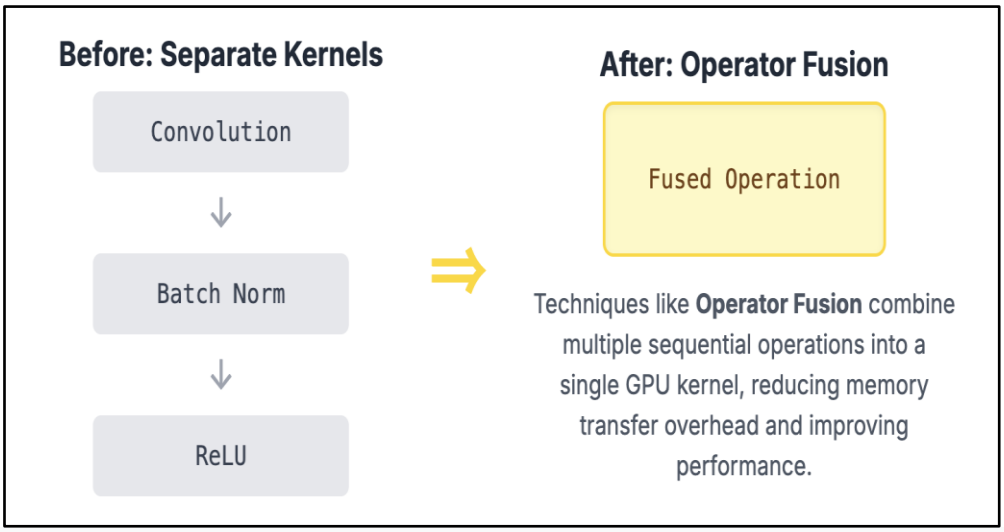

Fig 6. Compilers rewrite the model's computation graph for maximum hardware efficiency through operator fusion.

Publisher: SARC Publisher

Memory Layout Optimization

Memory layout optimization rearranges the storage pattern of tensors to enhance memory access patterns and cache usage. Such optimizations can effectively eliminate memory latency and bandwidth usage, especially on hardware with complex memory hierarchies.

The selection of appropriate tensor memory layouts is crucial to the performance of heterogeneous hardware architectures. Earlier deep learning systems typically employed standard memory layouts, such as NCHW (batch, channels, height, width) or NHWC (batch, height, width, channels), for convolution operations. However, optimal layouts can depend not only on hardware characteristics but also on the shapes of operations. The TensorFlow XLA compiler employs layout assignments that minimize costly transposition operations by considering the entire computation graph and propagating layout constraints to achieve optimal coherence between operators. This global optimization strategy contrasts with local layout choices determined by individual operators within conventional execution engines. Hardware-specific factors significantly influence the optimal layout choice. For example, current GPUs generally favor NHWC layouts that enhance cache locality for channel-based operations, while specific accelerators maximize memory bandwidth utilization with optimized NCHW layouts. Advanced compilers perform automatic layout conversion and padding to align memory access patterns with hardware vector units and cache line sizes (Chen, T. *et al.,* 2018).

Constant Folding and Propagation

Constant folding evaluates expressions involving only constant values at compile time instead of inference time. Constant propagation further extends the idea by replacing known constant values within expressions, thereby decreasing the computational costs even further during model execution.

These compiler optimization methods, initially designed for legacy programming languages, have been transformed to the particular structure of deep learning computational graphs. In contemporary compilers such as MLIR, constant folding is applied on the Static Single Assignment (SSA) form of the computational graph, detecting operations with all constants as inputs and substituting them with their calculated values at compile time. This optimization works exceptionally well for neural networks that have intricately initialized patterns, hard-coded normalization values, or architectural constants that pass through the model. Executing these optimizations within deep learning platforms requires careful consideration of numerical precision concerns, as constant folding on the host machine (typically using 64-bit arithmetic) can yield slightly different results from those obtained during runtime execution on accelerators (which often use 32-bit or lower precision). Constant aggressive propagation also interacts with other optimizations, such as operation fusion, sometimes demanding sophisticated coordination of optimization passes to get maximum benefit (Lattner, C. *et al.,* 2021).

Kernel Autotuning

Kernel autotuning is the automatic selection of optimal kernel implementations and parameters for particular hardware targets and workloads. It is the use of empirical performance measurement to determine the most efficient execution strategies for various operators and model components.

The increasing complexity of contemporary hardware accelerators has rendered analytical performance prediction progressively more demanding, leading to the adoption of empirical autotuning methodologies. Compilation systems based on MLIR utilize advanced autotuning systems that traverse parameterized code generation strategy spaces, empirically testing performance instead of relying on theoretical models. These compilation systems identify a search space that involves important performance factors such as tile sizes, loop unrolling factors, vectorization strategies, and memory promotion choices. The search in this space employs techniques ranging from brute force search for small parameter spaces to directed methods based on cost models or machine learning for larger spaces. Efficient autotuning involves a tradeoff between exploration time and deployment performance, where production systems in practice utilize transfer learning methods that leverage past tuning experiences to accelerate the optimization of new workloads. The performance gains from resulting optimizations can be dramatic, with best-optimized implementations usually performing 1.5-2× better than vendor-provided libraries for particular operations and targets (Lattner, C. *et al.,* 2021).

Compiler and Runtime Systems

Up-to-date AI compilers and runtimes provide comprehensive optimization pipelines that map

Publisher: SARC Publisher

high-level model representations into optimized executables tailored to a specific hardware platform. These systems employ several optimizations, including hardware-specific code generation, dataflow analysis and optimization, auto-parallelization, and memory allocation optimization.

The MLIR (Multi-Level Intermediate Representation) compiler infrastructure is a significant leap in AI compilation technology, giving a uniform framework for expressing and rewriting computations at all levels of abstraction. It facilitates a progressive lowering strategy from high-level operator graphs via multiple domain-specific representations to hardware-specific code. At every level, various optimization strategies become relevant—graph-level optimizations such as operation fusion and algebraic simplification at higher levels, loop and memory optimizations at mid levels, and instruction selection and register allocation at lower levels. This multi-level strategy obviates a core problem in deep learning compilation: the semantic gap between high-level frameworks and low-level hardware. By exposing proper abstractions at each level, MLIR enables framework developers and hardware providers to integrate their optimizations into a unified compilation pipeline, allowing optimized applications to be deployed across a more heterogeneous hardware environment that encompasses CPUs, GPUs, and specialized AI accelerators (Lattner, C. *et al.*, 2021).

Table 2: Runtime Optimizations for AI Model Deployment (Chen, T. *et al.*, 2018; Lattner, C. *et al.*, 2021)

Optimization Technique	Primary Performance Impact	Implementation Complexity	Hardware Dependency	Memory Impact
Operator Fusion	2-3× reduced memory bandwidth	Medium-High	Very High	Reduces transfers
Memory Layout Optimization	Improved cache utilization	Medium	High	Reduces fragmentation
Constant Folding/Propagation	Reduced computation at runtime	Low	Low	Minimal
Kernel Autotuning	1.5-2× performance vs vendor libraries	High	Very High	Varies
MLIR Compiler Framework	Multi-level optimization	Very High	Medium	Optimizes allocation

SYSTEM-LEVEL OPTIMIZATION TECHNIQUES

System-level optimizations target the deployment setting and user behavior patterns, converting computational efficiency into enhanced user experiences.

Dynamic Batching

Dynamic batching combines multiple incoming inference requests into a single batch to amortize the overhead cost and optimize hardware utilization. Dynamic batching is well-suited for services with non-uniform request rates, allowing for adaptive optimization of throughput in response to demand patterns.

State-of-the-art deep learning deployment systems increasingly employ dynamic batching to manage the intrinsic trade-off between efficiency and latency in serving contexts. The Triton Inference Server supports adaptive batching strategies that automatically adjust batch construction policies from arrival patterns and service-level requirements. In contrast to static batching during training, such systems must make decisions in real-time between hardware usage and user-perceived latency. Empirical results show that dynamic batching can increase throughput by up to four times for typical model architectures at acceptable latency profiles—however, the complexity of the implementation increases for variable-length inputs, which are common in NLP tasks. Padding and masking techniques must be intricately planned, especially for variable-length inputs, to minimize wasted computation. Complex production systems employ multi-level batching approaches that integrate conventional data-parallel batching with model-parallel execution to optimize hardware utilization across multiple dimensions simultaneously (Dao, T. *et al.*, 2022).

Publisher: SARC Publisher

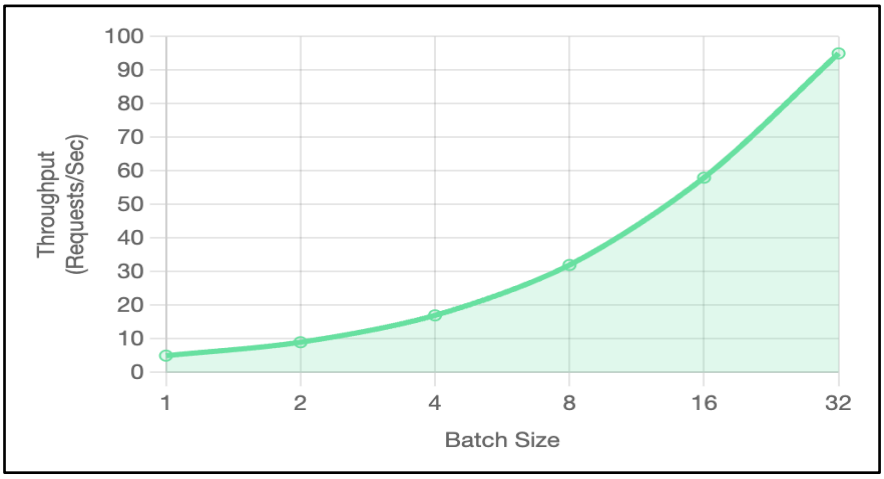

Fig 7. Increased GPU utilization and lowered the cost per request through dynamic batching.

KV-Cache Reuse

For transformer models, KV-cache reuse stores the key and value tensors from a previously performed forward pass to avoid redundant computation during sequential output generation. This reuse, enabled by KV-cache, is particularly useful in text generation applications where tokens are generated incrementally.

The computational pattern of transformer autoregressive decoding offers exceptional optimization opportunities using key-value cache management. DeepSpeed's design shows that the KV-cache design can drop the per-token generation complexity from $O(n^2)$ to $O(n)$, where n is the sequence length. The gains are especially valuable for long-form generation tasks, where naive methods would require redoubling attention computation for past tokens. Design considerations include memory layout optimization (with interleaved storage of keys and values often being faster than separate storage), cache invalidation strategies for interactive use, and system integration through batching. KV-caches' memory footprint linearly depends on sequence length and model size, imposing severe limitations for large models with long contexts (Narayanan, D. *et al.*, 2021).

Paged Attention

Paged attention mechanisms optimize memory management within large language models by applying virtual memory-like paging systems to attention operations. Paged attention enables the efficient processing of lengthy sequences and context windows that can otherwise not fit into available GPU memory.

The vLLM system introduces paged attention, utilizing virtual memory concepts to enhance transformer attention computation. This design divides KV-caches into fixed-size pages, effectively controlled using page tables, providing multiple critical abilities: (1) memory sharing at a fine-grained level between requests via page-level reference counting, (2) selective reclamation of memory via page replacement policies, and (3) sequence lengths over physical GPU memory via page swapping. The execution involves careful consideration of page size choice, weighing memory fragmentation against management overhead. Production systems describe 2-3× more extended context support than traditional techniques with interactive response times (Dao, T. *et al.*, 2022).

Request Coalescing

Request coalescing detects and merges similar inference requests, enabling computation sharing and reducing total resource requirements. The method is especially valuable for services that get many semantically equivalent queries in proximity in time.

Request coalescing achieves efficiency gains beyond trivial batching by identifying opportunities for computation sharing between semantically equivalent requests. The ZeRO-Infinity system demonstrates that identifying common computation patterns between requests can result in throughput gains of 20-50% beyond normal batching for heavily similar request workloads. Implementation approaches range from strict prefix matching for language generation to approximate approaches using embedding similarity for semantic coalescence. Technical realization requires advanced computation graph analysis and dynamic execution plans to effectively branch at points of divergence while

Publisher: SARC Publisher

maintaining logical isolation among requests (Narayanan, D. *et al.*, 2021).

Model Routing and Cascading

Intelligent routing of models sends incoming queries to correspondingly sized models based on question complexity, reserving the larger, more powerful models for complex tasks while running less complex queries with more efficient models. This methodology maximizes system-wide resource utilization without sacrificing response quality over a range of query types.

Model cascading employs incremental evaluation tactics that trade off between computational power and response quality. PipeDream's model routing system illustrates that efficient implementation necessitates three components: (1) estimation of request complexity via heuristics or learned models, (2) model selection policies with a trade-off between quality and resource availability, and (3) mechanisms for confidence assessment that decide when to switch to more powerful models. Production systems typically employ query-type-specific models for frequent queries, in conjunction with general models for handling complex cases. This strategy can reduce computational costs by 40-70% while maintaining quality comparable to that of always utilizing the most potent model (Narayanan, D. *et al.*, 2021).

Table 3: System-Level Optimization Techniques for AI Inference (Dao, T. *et al.*, 2022; Narayanan, D. *et al.*, 2021)

Technique	Primary Benefit	Resource Impact	Implementation Complexity
Dynamic Batching	Throughput optimization	Increases memory usage during peaks	High
KV-Cache Reuse	Reduces generation complexity from $O(n^2)$ to $O(n)$	Linear memory growth with sequence length	Medium
Paged Attention	Extends context window capabilities	Manages memory through virtual paging	High
Request Coalescing	20-50% throughput gain beyond standard batching	Minimal additional resources	Very High
Model Routing/Cascading	40-70% reduction in computational costs	Varies based on model suite	High

EVALUATION FRAMEWORK FOR OPTIMIZATION DECISIONS

Practitioners must consider multiple factors when choosing and implementing post-training optimizations. Below is a structured optimization decision framework:

Key Performance Indicators

Measurement of post-training optimizations should have a rich measurement framework that captures the multidimensional effects of each method. The MLPerf Inference Benchmark has become an industry-standard benchmark for comparing inference on various hardware platforms and optimization methods. The industry-standard benchmark specifies reference implementations and evaluation procedures for primary tasks such as image classification, object detection, medical imaging, speech recognition, and natural language processing. The benchmark utilizes task-specific quality targets (e.g., 99% FP32 accuracy for image classification) to require optimizations to achieve acceptable performance while benchmarking latency and throughput across various scenario conditions. The MLPerf approach differentiates between server scenarios (prioritizing throughput at high load) and offline scenarios (prioritizing batch processing efficiency), acknowledging that optimization goals vary by deployment environment. Subsequent benchmark iterations have incorporated power measurement, enabling the comparison of energy efficiency in growing criticality for large-scale deployments, where electricity usage is a significant operating expense (MLCommons).

In addition to standardized benchmarking, practitioners use application-specific analysis models that elicit domain-relevant measurements. Interactive applications also have tail latency (i.e., the 95th or 99th percentile) as a more critical metric than average latency, because deviant requests in this case can be particularly harmful to the user experience. The use of memory has become an assessment of peak and steady-state requirements, especially significant to transformer-based models, where attention mechanisms can generate large temporary memory loads. Cost modeling increasingly integrates infrastructure

Publisher: SARC Publisher

expenses with operational metrics, enabling ROI-driven optimization decisions that consider both technical performance and business impact. Most organizations have continuous benchmarking pipelines that compare optimization methods with production loads, providing empirical evidence of effectiveness in real-world settings rather than relying on artificial benchmarks (MLCommons).

Trade-off Analysis

Optimization choices always come with a compromise among these metrics. For example, decreased precision quantization reduces memory and enhances throughput, but can affect accuracy. Over-pruning decreases the model size at the expense of accuracy on complicated instances. Dynamic batching also improves throughput at the expense of unit request latency. Larger KV caches enhance generation quality but increase memory demand.

The optimization trade-off analysis has progressed beyond straightforward accuracy-efficiency comparisons to more complex multi-objective evaluation systems. Recent work on model compression reveals that the relationship between model size and performance actually follows a power law, rather than a linear trend, with early compression (to 70-80% of the original size) being commonly accurate. In contrast, stronger compression shows an accelerating breakdown. This realization has given rise to adaptive optimization approaches that utilize multiple techniques on different model components, as determined by sensitivity analysis. The Pareto frontier method for visualizing and analyzing provides practitioners with the means to identify optimization configurations that indicate the best trade-offs, rather than suboptimal ones. For sequence models, the trade-off between generation quality and computational efficiency involves more than mere accuracy metrics; it also encompasses response diversity, factual coherence, and alignment with human preferences (Dantas, P. V. *et al.*, 2024).

Interaction effects among optimization methods introduce significant complexity to trade-off analysis. Computational graph optimization methods, such as operator fusion, can reduce the efficiency of subsequent pruning by producing fused operators that are not partially removable. Conversely, quantization and distillation often demonstrate synergistic effects, with distilled models showing greater resilience to precision reduction. The sequencing of optimization techniques can significantly impact outcomes, leading to the development of specialized optimization pipelines tailored to specific model architectures and deployment targets. Current methods increasingly utilize neural architecture search methods that simultaneously optimize model design and post-training strategies, addressing the entire pipeline as an end-to-end optimization issue instead of dividing the design of the architecture from optimization at deployment (Dantas, P. V. *et al.*, 2024). This integrated method generates more efficient solutions but requires a significant amount of computational resources for exploration.

Deployment Considerations

In addition to performance metrics, practitioners must also consider hardware compatibility, monitoring needs, versioning and rollback plans, and processes to prevent optimizations from compromising model reliability and safety.

The heterogeneity of deployment environments adds enormous complexity to optimization choices. The results of the MLPerf Inference benchmark illustrate the dramatic variation of optimization performance across hardware platforms, with methods that deliver remarkable gains on GPUs under certain circumstances offering little gain on CPUs or specialized accelerators. Production deployment strategies are increasingly employing hardware-aware optimization selection, which automatically identifies and applies the most effective techniques for each target platform through systematic benchmarking and configuration management. This approach requires sophisticated deployment pipelines that support multiple optimization profiles for the same base model, selecting the appropriate configurations based on the characteristics of the deployment environment. The sophistication in having several optimized variants requires strong versioning schemes that monitor model weights as well as optimization settings to provide reproducibility and allow systematic comparisons between approaches (MLCommons).

The working aspects of optimizing model deployment go beyond performance to encompass robustness, monitoring, and safety assurances. Experimental work in firm model compression illustrates that naively optimized models tend to be more susceptible to adversarial examples and distribution changes than their unoptimized versions. This phenomenon has given rise to

Publisher: SARC Publisher

specialized optimization methods that actively maintain robustness properties at the expense of less efficiency gain. Monitoring for optimized models requires increased telemetry, in addition to basic accuracy measures, such as behavioral consistency checks to detect unforeseen changes in model output that may elude aggregate performance metrics. Optimized model deployment strategies are increasingly utilizing progressive rollout strategies, which incrementally grow traffic allocation while monitoring both performance metrics and behavioral consistency indicators. Safety aspects for optimal models involve evaluating uncertainty calibration and handling edge cases, ensuring that efficiency improvements do not result in higher risk in necessary usage (Dantas, P. V. *et al.*, 2024).

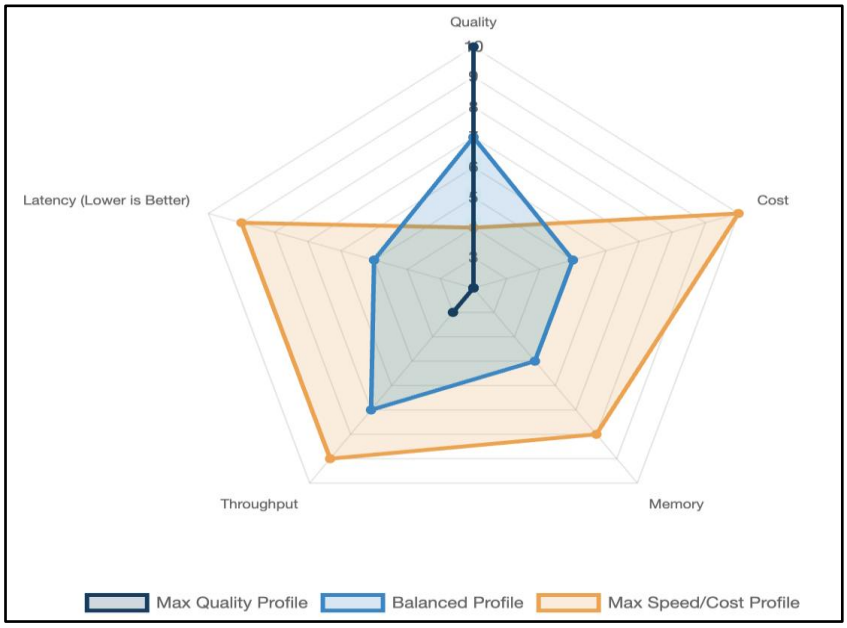

Fig. 8. Balanced trade-off profile for model deployed at scale

Table 4: Performance-Quality Trade-offs in Post-Training Optimization Techniques (MLCommons; Dantas, P. V. *et al.*, 2024)

Optimization Technique	Performance Impact	Memory Impact	Quality Impact	Deployment Complexity
Post-Training Quantization	Throughput	Memory Usage	Accuracy (minimal at 8-bit)	Medium
Model Pruning	Inference Speed	-Model Size	Complex Example Performance	Medium
Dynamic Batching	Throughput	Peak Memory	Latency Variation	High
KV-Cache Reuse	Generation Speed	Memory Usage	Generation Quality	Medium
Paged Attention	Context Length	Memory Management	Quality	High
Model Routing/Cascading	Resource Efficiency	Memory (varies)	Quality (task-dependent)	Very High

CONCLUSION

The factor of post-training optimization is a key to implementing AI models on a large scale. Optimization at the model, runtime, and system levels will help practitioners improve the effectiveness and efficiency of AI systems without affecting their primary functionality. The proposed framework will provide a systematic method for navigating the complex terrain of post-training optimization and assist in informed decision-making, considering quality, performance, resource usage, and cost. These approaches will become increasingly significant as AI systems become larger and more complex, enabling efficient and sustainable deployment. Future research directions include automated optimization selection, hardware-aware neural architecture search, and integrating optimization methods throughout the AI deployment stack. Their further development will enable the creation of more

Publisher: SARC Publisher

capable AI systems and control the costs of their calculations and environmental impact.

REFERENCES

1. Ahmed, A., Di, S., Cappello, F., Liu, Z., Han, J., & Anwar, A. "Systematic evaluation of optimization techniques for long-context language models." *arXiv preprint arXiv:2508.00305* (2025).
2. Sinha, S., & Lee, Y. M. "Challenges with developing and deploying AI models and applications in industrial systems." *Discover Artificial Intelligence* 4.1 (2024): 55.
3. Liang, T., Glossner, J., Wang, L., Shi, S., & Zhang, X. "Pruning and quantization for deep neural network acceleration: A survey." *Neurocomputing* 461 (2021): 370-403.
4. Houlsby, N., Giurgiu, A., Jastrzebski, S., Morrone, B., De Laroussilhe, Q., Gesmundo, A., ... & Gelly, S. "Parameter-efficient transfer learning for NLP." *International conference on machine learning*. PMLR, (2019).
5. Chen, T., Moreau, T., Jiang, Z., Zheng, L., Yan, E., Shen, H., ... & Krishnamurthy, A. "{TVM}: An automated {End-to-End} optimizing compiler for deep learning." *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. (2018).
6. Lattner, C., Amini, M., Bondhugula, U., Cohen, A., Davis, A., Pienaar, J., ... & Zinenko, O. "MLIR: Scaling compiler infrastructure for domain specific computation." *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, (2021).
7. Dao, T., Fu, D., Ermon, S., Rudra, A., & Ré, C. "Flashattention: Fast and memory-efficient exact attention with io-awareness." *Advances in neural information processing systems* 35 (2022): 16344-16359.
8. Narayanan, D., Phanishayee, A., Shi, K., Chen, X., & Zaharia, M. "Memory-efficient pipeline-parallel dnn training." *International Conference on Machine Learning*. PMLR, (2021).
9. MLCommons, "MLPerf Inference: Datacenter,".
10. Dantas, P. V., Da Silva, W. S., Cordeiro, L. C., & Carvalho, C. B. "A comprehensive review of model compression techniques in machine learning." *Applied Intelligence* 54.22 (2024): 11804-11844.

Source of support: Nil; **Conflict of interest:** Nil.

Cite this article as:
Kumar, R. " Post-Training Optimization Techniques for AI Models: A Comprehensive Framework." *Sarcouncil Journal of Engineering and Computer Sciences* 4.11 (2025): pp 231-243.

Publisher: SARC Publisher