Ahead Of Its Time: Exploring AOT vs JIT Compilation Strategies in Deep Learning Frameworks

Shane Dirksen*
Department of Computer Science
University of California, Santa Barbara
Santa Barbara, California
shanedirksen@ucsb.edu

Ron Kibel*
Department of Computer Science
University of California, Santa Barbara
Santa Barbara, California
rkibel@ucsb.edu

Abstract

In deep learning, static graphs (as used in ONNX Runtime and TensorRT) predefine the entire computation flow prior to execution, which can allow for aggressive compiler optimization but can limit the runtime flexibility (changing input size, control flow, randomness). Dynamic graphs, on the other hand (like those in PyTorch's "eager mode"), are built on the fly, which can offer easier debugging and adaptable behavior but come at a cost in performance. Understanding this tradeoff is incredibly relevant as frameworks converge toward hybrid systems that balance adaptability and efficiency.

Our project, AheadOfItsTime, explores the performance and flexibility tradeoffs between Ahead-of-Time (AOT) and Just-in-Time (JIT) compilation strategies in modern DL frameworks and how these approaches interact with static and dynamic computation graphs. Using CNN and RNN benchmarks, we measure inference latency, memory usage, and compile-time overhead across multiple DL runtimes to quantify how AOT and JIT compilation influence runtime efficiency for common workloads. Our results demonstrate that the benefits of compilation are highly architecture-dependent and sensitive to batch size. For CNNs, AOT runtimes are capable of achieving significant speedup in small batches, yet suffer performance inversion as the batch size increases. In contrast RNNs largely resist optimization, regardless of the runtime involved. We conclude that no single execution strategy is universally superior; TensorRT is optimal for latency-sensitive CNN inference, TorchScript is preferable for high-throughput batched CNNs, and PyTorch Eager remains the most effective solution for recurrent workloads.

Index Terms

Deep Learning, Just-In-Time compilation, Ahead-Of-Time compilation, Benchmark testing

I. Introduction

Deploying deep learning models in production means trading off raw performance against flexibility and engineering effort. Modern systems rarely run "vanilla" framework code. Instead, they rely on compilation stacks such as PyTorch 2.0's torch.compile, ONNX Runtime, and NVIDIA TensorRT, or on more traditional eager execution paths like standard PyTorch [1][2][4].

Ahead-of-time systems treat the model as a mostly static computation graph and perform whole-graph optimizations such as operator fusion, memory layout planning, and kernel selection before inference. TensorRT and ONNX Runtime are typical examples: they take a trained model, run a sequence of graph and kernel-level optimizations, and emit an engine specialized for a target device[2][4]. This usually improves throughput and latency, but pushes users toward fixed input shapes, longer compile times, and less transparent debugging. In contrast, dynamic "eager" execution as in standard PyTorch keeps operations in Python, which supports arbitrary control flow and rapid iteration but limits cross-operation optimization and keeps the interpreter in the hot path[1]. PyTorch 2.0's torch.compile tries to recover some of the benefits of compilation by capturing graphs out of eager programs via TorchDynamo and generating fused kernels with TorchInductor.

There is already benchmarking work around these systems, but it tends to emphasize specific slices of the design space. Framework and compiler papers often report speedups on small sets of canonical CNN and transformer models (e.g., ResNet, BERT) for particular hardware and batch sizes [1][5]. Larger suites such as TorchBench [3] cover many models and backends, but they are designed as regression and performance-tracking tools rather than controlled studies of how AOT vs JIT vs hybrid strategies behave across model families and deployment constraints. As a result, practitioners still end up making many deployment decisions by trial and error, especially for recurrent models or latency-sensitive services where compilation time and cold-start behavior matter.

Code and data available at: https://github.com/rkibel/AheadOfItsTime
* Both authors contributed equally to this research.

This project takes a deliberately small but varied slice through that space. We benchmark five execution engines (PyTorch eager, TorchScript, torch.compile, ONNX Runtime, TensorRT) across four model architectures (LeNet-5, ResNet-18, LSTM, GRU) spanning convolutional and recurrent workloads. For each model-engine pair, we measure not only steady-state latency but also GPU memory consumption, compilation overhead, and energy use across batch sizes 1, 8, 32, 128. Our aim is to characterize when compilation is actually worth its cost, how benefits differ between CNNs and RNNs, and what this implies for realistic deployment choices.

II. Methodology

We benchmark four models across five execution engines:

A. Models

- **LeNet-5 (MNIST, ∼60K parameters)**: Small CNN with a classic Conv–Pool–Conv–Pool–FC layout; simple, regular computation.
- **ResNet-18 (CIFAR-10, ∼11M parameters)**: Deeper CNN with residual connections and batch normalization; representative of modern vision models.
- **Bidirectional LSTM (IMDB sentiment, ∼2M parameters, up to 512 tokens)**: Recurrent model with dynamic unrolling and irregular memory access over hidden states.
- **Stacked GRU (WikiText-2 language modeling, ∼5M parameters)**: Multi-layer GRU network emphasizing sequential processing and hidden-state reuse.

B. Execution engines

- **PyTorch eager**: Baseline dynamic execution with no cross-op graph optimization; operations remain in Python.
- **TorchScript**: JIT compilation via tracing/scripting with optimization passes over the IR.
- **torch.compile**: Hybrid mode using TorchDynamo to capture graphs from eager code and TorchInductor to generate fused kernels.
- **ONNX Runtime**: AOT execution via ONNX export, graph optimizations, and the CUDA execution provider.
- **TensorRT**: NVIDIA-specific AOT compiler with aggressive layer fusion, kernel selection, and precision calibration for low-latency inference.

C. Experimental setup

Each model-framework-batch size combination undergoes 100 warmup iterations followed by 1000 inference iterations for latency measurement. Latency measurements use CUDA event synchronization, extracting median, P95, and P99 values. Memory profiling captures peak and average GPU consumption over 100 iterations. Energy profiling uses NVIDIA Management Library (NVML) to sample GPU power every 10ms during inference, calculating average watts and inferences per joule. Batch sizes [1, 8, 32, 128] span single-request to mini-batch scenarios.

All models use identical warmup procedures, random seeds, and CUDA contexts. A unified conversion pipeline validates numerical equivalence across frameworks. Benchmarks execute on NVIDIA GeForce RTX 3090 Ti, CUDA 12.1, PyTorch 2.5.1+cu121.

III. Results

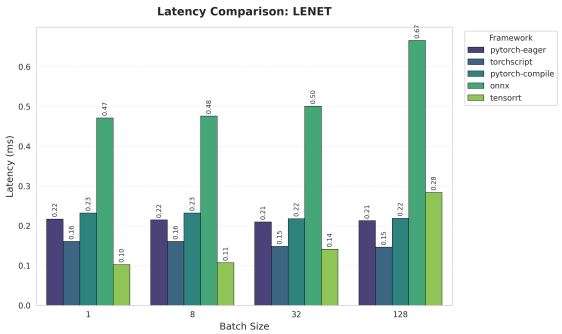
A. CNN Performance

For CNNs, compilation delivers consistent performance gains at small batch sizes. TensorRT achieves 2.12x speedup on LeNet-5 and 2.00x on ResNet-18 versus PyTorch eager at batch size 1. TorchScript delivers 1.32x on LeNet and 1.87x on ResNet-18. torch.compile achieves 1.26x on ResNet-18 but slows to 0.96x on LeNet.

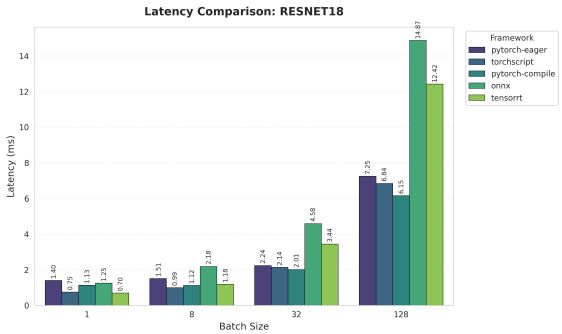
CNNs present regular computation patterns with predictable data flow, making them amenable to aggressive operator fusion. TensorRT's kernel-level optimizations combine Conv+ReLU+Pool operations into single CUDA kernels, eliminating intermediate memory writes. This accounts for the significant batch size 1 speedups.

At batch size 128, however, performance rankings reverse. TensorRT drops to 0.73x on LeNet (37% slower than eager) and 0.57x on ResNet-18 (43% slower). TorchScript maintains 1.42x on LeNet and 1.06x on ResNet-18. This reversal reveals TensorRT's optimization target: minimizing kernel launch overhead for low-latency single-inference scenarios. At large batch sizes, TensorRT's aggressive fusion reduces parallelism opportunities, preventing GPU saturation.

ONNX Runtime underperforms despite AOT compilation. At batch size 1, ONNX is 0.45x (2.2x slower) on LeNet and 1.11x on ResNet-18. At batch size 128, ONNX drops to 0.31x on LeNet and 0.49x on ResNet-18. Framework-agnostic operators sacrifice PyTorch's vendor-optimized CUDA kernels, demonstrating that portability and performance are opposing forces.



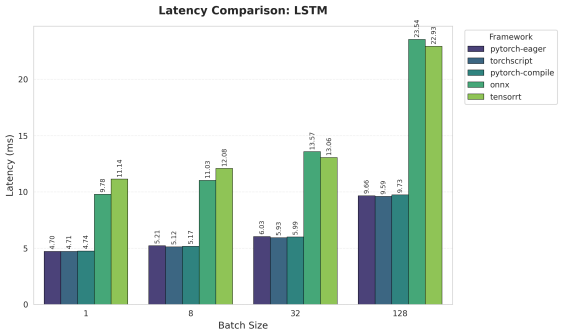
(a) Inference latency for LeNet



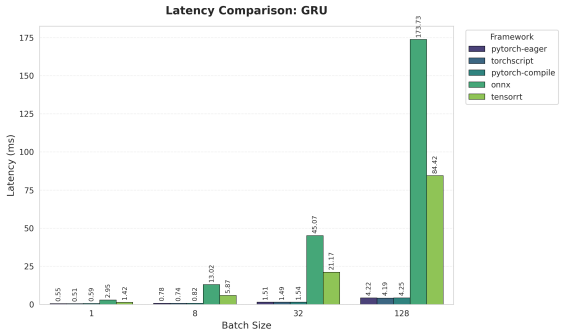
(b) Inference latency for ResNet

Fig. 1: Inference latency for CNN architectures.

B. RNN Performance



(a) Inference latency for LSTM



(b) Inference latency for GRU

Fig. 2: Inference latency for RNN architectures.

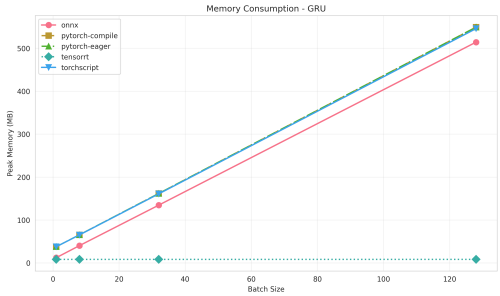
RNNs resist compilation optimization across all frameworks and batch sizes. At batch size 1, speedups range from 0.98x to 1.18x. TorchScript achieves 1.18x on GRU and 1.03x on LSTM. torch.compile delivers 1.02x on GRU and 0.98x on LSTM. TensorRT achieves 0.42x on GRU and 0.44x on LSTM (2.3-2.4x slower than eager).

The fundamental difference from CNNs is sequential computation with hidden state dependencies. Each RNN timestep depends on the previous timestep's output, forcing sequential execution that cannot be parallelized. Graph optimizations like operator fusion fail because they assume independent, parallelizable operations.

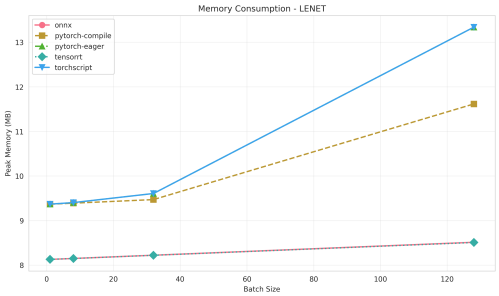
At batch size 128, AOT framework performance collapses. ONNX drops to 0.02x on GRU (50x slower) and 0.41x on LSTM. TensorRT drops to 0.05x on GRU (20x slower) and 0.42x on LSTM. TorchScript and torch.compile remain near parity (1.00x and 0.99x). TensorRT's aggressive fusion backfires on sequential workloads, where static memory planning and kernel fusion create overhead that dominates any optimization benefit.

This result demonstrates that compilation strategy effectiveness is architecture-dependent. Implementation details matter as much as the overall strategy: torch.compile shows JIT can match AOT on CNNs (1.26x on ResNet-18), but TorchScript and torch.compile diverge on RNNs where neither provides meaningful speedup.

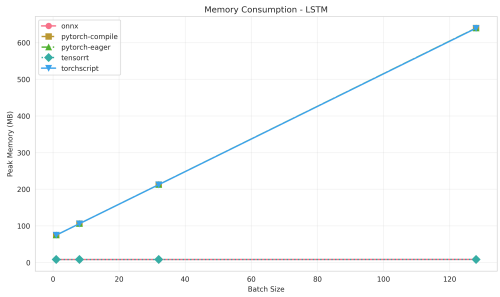
C. Memory Consumption



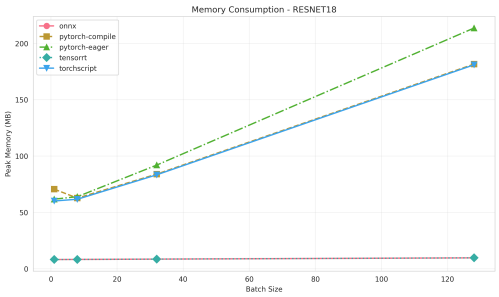
(a) GRU



(b) LeNet



(c) LSTM

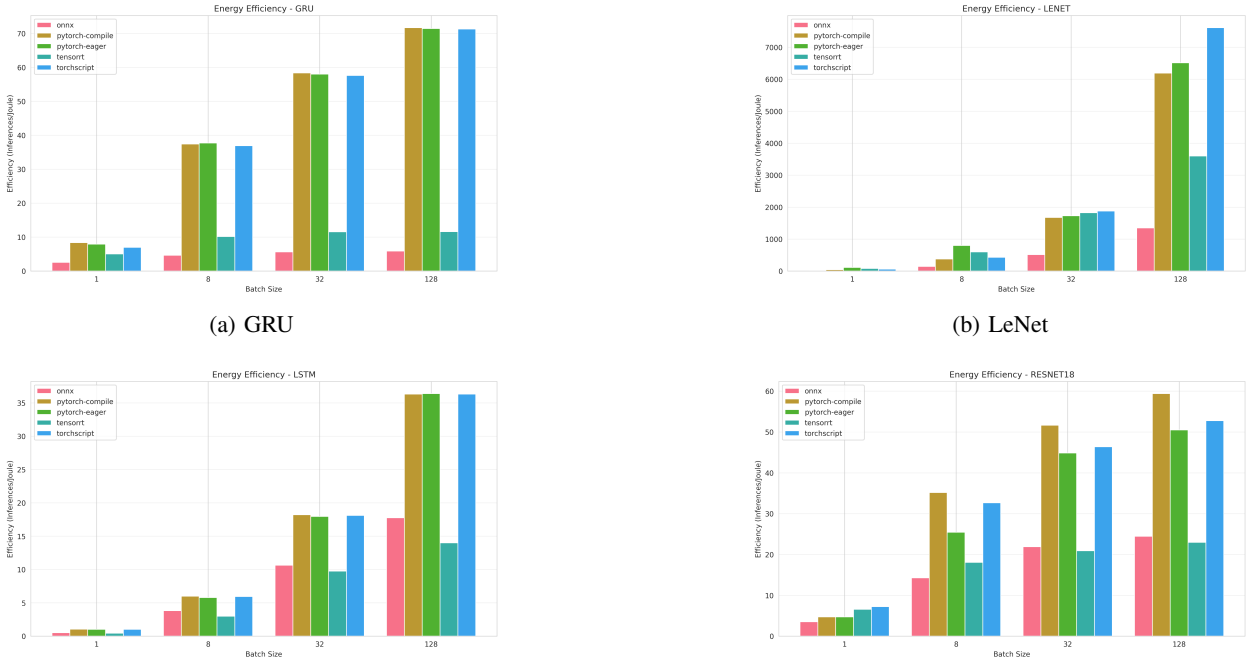


(d) ResNet18

Fig. 3: Peak memory consumption across all model architectures.

ONNX Runtime and TensorRT reduce memory consumption dramatically. On RNNs at batch size 1, peak memory drops from 75MB (eager, LSTM) to 8MB (ONNX/TensorRT), an 89% reduction. On GRU, memory drops from 38MB to 8-12MB, a 68-79% reduction. On CNNs, memory reductions are smaller. ResNet-18 drops from 62MB (eager) to 8MB (ONNX/TensorRT), an 87% reduction. LeNet shows minimal difference (9.37MB vs 8.13MB). torch.compile increases memory on ResNet-18 (71MB vs 62MB eager), a 15% increase, suggesting graph capture overhead.

D. Energy Efficiency



(a) GRU (b) LeNet

(c) LSTM (d) ResNet18

Fig. 4: Energy consumption across all model architectures.

Energy efficiency correlates with latency for most frameworks. On LeNet at batch size 1, eager achieves 89.4 inferences/joule, TensorRT 77.1, TorchScript 52.4, torch.compile 44.8, and ONNX 18.7. TensorRT's lower efficiency, despite 2x speedup, indicates higher power draw during optimized execution. However, for RNNs, energy efficiency is low across all frameworks. LSTM eager achieves 1.0 at batch size 1, dropping to 0.5 for ONNX/TensorRT despite their memory advantages.

Moreover, as batch size increases, efficiency increases across all frameworks. We conclude that batching amortizes fixed overhead.

IV. Analysis

Framework selection must be driven by model architecture and batch size. For CNNs at batch size 1, TensorRT delivers 2x speedups. At batch size 128, TensorRT underperforms eager by 27-43%, making TorchScript the better choice for batched CNN inference (1.06-1.42x speedup maintained).

For RNNs, compilation delivers minimal benefit ($< 3\%$ speedup for TorchScript/torch.compile at batch size 1) or severe degradation (20-50x slower for TensorRT/ONNX at batch size 128). PyTorch eager mode is the pragmatic choice for recurrent architectures. Exception: memory-constrained deployments where ONNX/TensorRT's 68-89% memory reduction enables deployment despite latency penalty.

ONNX Runtime underperforms PyTorch eager across all models and batch sizes, challenging assumptions about AOT compilation benefits. On CNNs at batch size 1, ONNX is 0.45-1.11x versus eager. On RNNs, ONNX is 0.02-0.51x. Framework-agnostic operators lack PyTorch CUDA kernel optimizations, and graph optimization overhead outweighs benefits.

TensorRT's batch size sensitivity is critical for deployment decisions. Single-inference scenarios benefit from 2x speedup. Batched scenarios suffer 27-43% slowdown on CNNs and 20-50x slowdown on RNNs. Services with dynamic batching must benchmark at representative batch sizes.

torch.compile shows inconsistent performance. On ResNet-18, it matches or exceeds TorchScript (1.26x vs 1.87x at batch size 1). On LeNet, it underperforms eager (0.96x). Hybrid compilation strategies show marginal benefits over specialized frameworks.

Energy efficiency varies by batch size and framework. At batch size 1, TorchScript achieves best efficiency on ResNet-18 despite slower latency than TensorRT. At batch size 128, batching amortizes fixed overhead and efficiency increases 100-1000x across all frameworks. Framework selection must consider typical batch size distribution for energy cost optimization.

Memory-latency tradeoffs on RNNs (68-89% memory reduction, 2-50x latency penalty for ONNX/TensorRT) are viable only for specific deployment constraints where memory is the limiting factor.

V. Conclusion

Compilation effectiveness is architecture-dependent and batch-size-dependent. CNNs benefit from compilation at batch size 1 (TensorRT 2x speedup) but not at batch size 128 (TensorRT 0.57-0.73x). RNNs resist optimization across all batch sizes (3% speedup for JIT, 2-50x slowdown for AOT).

ONNX Runtime's underperformance despite AOT compilation indicates practitioners should empirically validate compilation benefits rather than assume AOT superiority.

Deployment recommendations: For CNN inference at batch size 1, use TensorRT (2x speedup). For batched CNN inference (batch size 128), use TorchScript (1.06-1.42x speedup) as TensorRT underperforms. For RNN applications, use PyTorch eager as compilation overhead is not justified by marginal gains or introduces severe degradation. For memory-constrained RNN deployments, ONNX/TensorRT memory reduction (68-89%) may justify latency penalty.

torch.compile shows inconsistent performance across architectures and does not demonstrate clear superiority over TorchScript or specialized frameworks. For organizations using TorchScript, migration offers marginal benefit.

Future work should extend to transformer architectures (attention mechanisms suggest CNN-like compilation benefits, but quadratic sequence complexity introduces challenges), dynamic batching and variable-length sequences (real-world serving stresses static shape optimization), quantization interaction with compilation (INT8/FP16 standard in production), cross-platform portability (TPUs, edge devices), and system-level energy profiling (CPU, memory, I/O, not just GPU).

References

[1] Jason Ansel et al. "PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation". In: *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '24)*. ACM, 2024, pp. 929–947. DOI: 10.1145/3620665.3640366.

[2] ONNX Runtime developers. *ONNX Runtime: High-Performance Inference Engine*. https://onnxruntime.ai/. version x.y.z. 2021.

[3] Yueming Hao et al. "TorchBench: Benchmarking PyTorch with High API Surface Coverage". In: *arXiv preprint arXiv:2304.14226* (2023). GitHub repository: pytorch/benchmark.

[4] NVIDIA Corporation. *NVIDIA TensorRT: High-Performance Deep Learning Inference Optimizer and Runtime*. Developer Guide / whitepaper, version 10.2.0 (or other version as appropriate). NVIDIA Corporation. 2024. URL: https://docs.nvidia.com/deeplearning/tensorrt/developer-guide/index.html.

[5] Adrien Payong and Shaoni Mukherjee. "Optimize Production with PyTorch/TF, ONNX, TensorRT LiteRT". In: *DigitalOcean Community* (2025). Published October 3, 2025.