electronics

Article

Accelerating Deep Learning Inference: A Comparative Analysis of Modern Acceleration Frameworks

Ishrak Jahan Ratul *, Yuxiao Zhou and Kecheng Yang

Department of Computer Science, Texas State University, San Marcos, TX 78666, USA; yuxiao.zh@gmail.com (Y.Z.); yangk@txstate.edu (K.Y.)
* Correspondence: ishrakratul@txstate.edu

Abstract

Deep learning (DL) continues to play a pivotal role in a wide range of intelligent systems, including autonomous machines, smart surveillance, industrial automation, and portable healthcare technologies. These applications often demand low-latency inference and efficient resource utilization, especially when deployed on embedded or edge devices with limited computational capacity. As DL models become increasingly complex, selecting the right inference framework is essential to meeting performance and deployment goals. In this work, we conduct a comprehensive comparison of five widely adopted inference frameworks: PyTorch, ONNX Runtime, TensorRT, Apache TVM, and JAX. All experiments are performed on the NVIDIA Jetson AGX Orin platform, a high-performance computing solution tailored for edge artificial intelligence workloads. The evaluation considers several key performance metrics, including inference accuracy, inference time, throughput, memory usage, and power consumption. Each framework is tested using a wide range of convolutional and transformer models and analyzed in terms of deployment complexity, runtime efficiency, and hardware utilization. Our results show that certain frameworks offer superior inference speed and throughput, while others provide advantages in flexibility, portability, or ease of integration. We also observe meaningful differences in how each framework manages system memory and power under various load conditions. This study offers practical insights into the trade-offs associated with deploying DL inference on resource-constrained hardware.

Keywords: deep learning; PyTorch; real-time inference; TensorRT; JAX; Apache TVM

Academic Editors: Sergio Spanò, Gian Carlo Cardarilli and Luca Di Nunzio

Received: 16 June 2025
Revised: 19 July 2025
Accepted: 21 July 2025
Published: 25 July 2025

Citation: Ratul, I.J.; Zhou, Y.; Yang, K. Accelerating Deep Learning Inference: A Comparative Analysis of Modern Acceleration Frameworks. *Electronics* **2025**, *14*, 2977. https://doi.org/10.3390/electronics14152977

1. Introduction

In recent years, DL has experienced rapid growth and has emerged as one of the most effective techniques within the field of machine learning. This progress has fueled the integration of DL into a wide variety of applications, including computer vision, natural language processing (NLP), and autonomous control systems. Increasingly, these applications are being deployed not only on general-purpose computing platforms but also on embedded and edge systems that operate under strict power, size, and latency constraints.

Unlike traditional machine learning algorithms, which often suffer from overfitting or poor scalability in large data settings, modern deep neural networks (DNNs) can maintain high accuracy even with overparameterized architectures. For example, the parameter count in state-of-the-art image classification models has increased dramatically, from 61 million to more than 2.1 billion between 2013 and 2023, as tracked by the ImageNet leaderboard [1]. Similarly, in NLP, models like BERT [2] and its successors contain hundreds of millions of parameters, offering breakthrough performance while introducing significant computational overhead [3].

Although such large models achieve exceptional accuracy, their deployment comes at a cost. Training and especially inference can be slow, memory-intensive, and computationally expensive. This poses a major challenge in scenarios where inference must be conducted in real time while operating under limitations on size, weight, power, and cost. Embedded and edge platforms, such as those used in robotics, drones, mobile healthcare, or surveillance, must perform inference with limited processing power, memory bandwidth, and energy. These platforms often require instant responses, which makes the inefficiencies of standard DL frameworks a critical concern.

Consider autonomous vehicles as an example. They must process high-bandwidth sensor inputs such as LiDAR and video feeds in real time to ensure safe navigation. Similarly, video surveillance systems must detect incidents such as theft, fire, or physical conflict immediately, often without relying on remote cloud services. In these and similar scenarios, executing inference locally on embedded devices becomes essential due to requirements for low latency, reliability, and data privacy.

This gap between high-performing models and the constraints of real-world deployment environments has led to the development of various strategies to optimize inference. Approaches such as quantization [4], pruning [5], and neural architecture search (NAS) [6] aim to reduce the resource demand of models. Lightweight networks such as MobileNet [7] and SqueezeNet [8] are designed specifically for deployment under limited hardware resources. In addition, specialized hardware solutions including Google's TPU [9] and Intel's VPU [10] have been introduced to accelerate DL inference at low power consumption.

Despite these advances, widely used frameworks such as TensorFlow and PyTorch [11] are not inherently optimized for inference on embedded systems. To address this issue, several software-level optimization tools have emerged. For instance, NVIDIA developed TensorRT [12], a high-performance inference engine designed to transform trained models into highly efficient executables suitable for edge and automotive platforms.

In addition to TensorRT, other frameworks are increasingly used for inference acceleration. ONNX Runtime allows deployment across platforms using different execution providers [13]. Apache TVM optimizes and compiles models for specific hardware targets [14]. JAX supports high-performance computation through just-in-time compilation, although its deployment maturity on embedded platforms is still developing [15].

While many of these tools have demonstrated strong potential, there is a need for a consistent evaluation of their performance under realistic deployment conditions. The NVIDIA Jetson AGX Orin provides a modern, high-efficiency edge AI platform that integrates Ampere architecture GPUs with AI acceleration hardware [16]. This platform offers a timely opportunity to benchmark inference frameworks across critical performance dimensions.

In this study, we present a comparative evaluation of five prominent DL inference frameworks: PyTorch, ONNX Runtime, TensorRT, Apache TVM, and JAX. Our benchmarking is conducted on the Jetson AGX Orin platform, using representative models and measuring key metrics such as inference time, throughput, memory utilization, power consumption, and accuracy. The results offer insights for developers and researchers working on real-time AI applications in resource-constrained environments.

Contribution

This paper makes the following key contributions to the evaluation of DL inference performance on embedded systems:

- We conduct a comprehensive evaluation of five popular DL inference frameworks: PyTorch, ONNX Runtime, TensorRT, Apache TVM, and JAX, on the NVIDIA Jetson AGX Orin platform.

- The benchmarking covers inference time, throughput, system memory usage, power consumption, and prediction accuracy using a unified and consistent experimental setup. Both Top 1 and Top 5 classification accuracy are reported.
- All evaluations are performed using the ImageNet validation dataset comprising 50,000 images across 1000 classes, ensuring realistic and standardized performance comparison.
- The study includes both convolutional and transformer-based models, representing a diverse range of modern neural network architectures used in computer vision and general artificial intelligence.
- We present practical insights into the tradeoffs associated with each framework, offering guidance for selecting suitable inference solutions for embedded and resource-constrained deployments.
- To support reproducibility, we detail the complete benchmarking methodology, including model conversion workflows, optimization strategies, measurement protocols, and software configurations.

Organization

The subsequent sections of this paper are structured in the following manner: Section 3 offers a comprehensive introduction to PyTorch, ONNX Runtime, TensorRT, TVM, and JAX. Section 4 outlines the methodology used in our studies, which includes the models that were tested, the workflows that were followed, and the methods used to measure performance. The experimental findings and discussions may be found in Sections 5 and 6. Section 2 presented a comprehensive summary of the existing research. We conclude our article in Section 7.

2. Related Work

Numerous studies have investigated deep learning inference optimization and deployment on edge and embedded systems. Cheng et al. [17] surveyed compression and acceleration techniques such as quantization, pruning, distillation, and architecture search. Hao et al. [18] benchmarked edge AI models, demonstrating how deployment performance depends on the interplay among model structure, hardware architecture, and compiler technology.

In 2022, Shin and Kim [19] evaluated YOLO models using TensorRT on Jetson platforms, confirming real-time object detection suitability. Zhang et al. [20] assessed ONNX Runtime as a hardware-agnostic backend, emphasizing portability benefits despite limited edge-focused benchmarks. The Apache TVM compiler was introduced by Chen et al. [14], showcasing performance tuning across diverse hardware targets. Peng et al. [21] examined JAX's high-performance numerical computation on cloud and server environments, leaving its edge-inference behavior less explored. Ulker et al. [22] compared TensorFlow Lite and OpenVINO on Raspberry Pi and Jetson TX2, while Wortsman et al. [23] studied ensemble methods on high-end GPUs without embedded device focus.

More recent work continues this trend. Alqahtani et al. [24] performed extensive benchmarking of object detection models, including YOLOv8, on Jetson AGX Orin Nano and Raspberry Pi, highlighting trade-offs between accuracy, latency, and energy efficiency. Yeom and Kim [25] introduced UniForm, a vision-transformer variant optimized for edge devices like Jetson AGX Orin, achieving up to fivefold speed gains. Arya and Simmhan [26] evaluated large language model inference (2.7 B–32.8 B parameters) on Jetson Orin AGX, analyzing batch size, quantization, latency, throughput, and power, offering insight into the feasibility of edge-based LLMs.

Despite these advances, to the best of our knowledge, no prior work offers a comprehensive, multi-metric comparison across PyTorch (v2.3.0), ONNX Runtime (v1.17.1), TensorRT (v8.6.2.3), Apache TVM (v0.21.0), and JAX (v0.4.28) on the Jetson AGX Orin platform.

Our research fills this gap by benchmarking both convolutional and transformer models on ImageNet across inference time, throughput, memory, power, and Top-1/Top-5 accuracy.

3. Overview of Inference Frameworks

3.1. PyTorch

PyTorch [11] is an open-source machine learning library that facilitates moving from research prototyping to production deployment rapidly. It is primarily utilized as a DL platform in Python, offering exceptional flexibility and speed for research applications.

PyTorch allows for the manipulation of Tensors (multi-dimensional arrays) across both CPUs and GPUs, significantly speeding up computations. It offers a broad spectrum of tensor operations catering to diverse scientific computing needs, including both basic arithmetic and advanced linear algebra.

Unlike other frameworks where a neural network's architecture must be predefined and reused, PyTorch employs reverse-mode auto-differentiation. This method enables users to alter network behavior on the fly without substantial computational overhead.

Enhanced by acceleration libraries like Intel MKL and NVIDIA (cuDNN, NCCL), PyTorch performs efficiently across various network sizes. It is also optimized for memory usage, enabling the training of very large DL models without the memory constraints typical of other frameworks.

3.2. ONNX Runtime

ONNX Runtime [20] is a high-performance inference engine developed by Microsoft to support the Open Neural Network Exchange (ONNX) format. It is designed to maximize inference speed and portability across a wide range of hardware platforms and environments.

ONNX Runtime allows developers to deploy trained models from multiple frameworks such as PyTorch, TensorFlow, and scikit-learn, offering backend flexibility through various execution providers. These include CPU, CUDA GPU, TensorRT, DirectML, and OpenVINO, enabling seamless adaptation to different deployment targets. The modular architecture makes it possible to switch hardware acceleration paths with minimal code changes.

To improve performance, ONNX Runtime supports optimizations such as operator fusion, constant folding, graph pruning, and quantization. It also integrates with model acceleration tools like Intel Neural Compressor and NVIDIA TensorRT to further optimize inference. ONNX Runtime is widely used in production-scale deployments for its efficient memory usage and predictable latency characteristics. It is especially well-suited for inference scenarios requiring cross-platform consistency and low deployment overhead.

3.3. TensorRT

TensorRT is a high-performance DL inference SDK, part of the NVIDIA CUDA X AI Kit. It includes an inference optimizer and runtime that achieves low latency and high throughput.

TensorRT enhances DL model performance through six optimization strategies: (1) Weight and activation precision calibration: optimizes model performance by quantizing to 8-bit integers while maintaining accuracy. (2) Layer and tensor fusion: consolidates nodes within a kernel to improve GPU memory use and bandwidth. (3) Kernel auto-tuning: optimizes based on the GPU platform to choose the best layers, algorithms, and batch sizes. (4) Dynamic tensor memory: efficiently allocates memory only when needed, reducing consumption and allocation overhead. (5) Multi-stream execution: processes multiple input streams concurrently. (6) Time fusion: optimizes RNNs by dynamically generating kernels across time steps.

TensorRT supports a broad spectrum of AI applications, from computer vision and automatic speech recognition to natural language understanding and text-to-speech. It provides ready-to-deploy inference engines for diverse applications, including autonomous driving and real-time video analytics, ensuring efficient real-time inference on edge devices and in IoT scenarios.

3.4. Apache TVM

Apache TVM [14] is an open-source DL compiler stack that enables the deployment of machine learning models across a diverse set of hardware backends. TVM translates high-level model representations into optimized code tailored for the target device, including CPUs, GPUs, and specialized accelerators.

TVM provides end-to-end compilation from frameworks such as PyTorch, TensorFlow, and Keras. It performs automated graph-level and tensor-level optimizations including operator fusion, memory reuse, loop unrolling, and layout transformation. These features are critical for reducing inference latency and memory consumption, especially on embedded devices.

A key feature of TVM is its AutoTVM module, which uses machine learning-based cost models to perform hardware-aware tuning. This allows TVM to search for optimal schedules that balance computation and memory usage for a specific target. The resulting compiled models are highly efficient and portable.

Due to its flexibility and performance, TVM is often used in research and production environments where fine-grained control over deployment is essential. It continues to be extended to support microcontrollers, NPUs, and custom hardware accelerators.

3.5. JAX

JAX [21] is a numerical computing library developed by Google Research, designed for high-performance machine learning research. It provides composable function transformations such as automatic differentiation, vectorization, and just-in-time compilation using the Accelerated Linear Algebra (XLA) compiler.

Unlike traditional DL frameworks, JAX emphasizes functional programming paradigms and offers seamless interoperability with NumPy. It supports efficient large-scale numerical computation on both CPUs and GPUs, and is particularly well-suited for gradient-based optimization and scientific simulations.

JAX compiles Python functions into optimized machine code for target devices using XLA. This compilation not only improves performance but also reduces runtime overhead. It is capable of automatically parallelizing code across multiple devices and supports distributed training through libraries such as Flax and Haiku.

Although JAX is primarily used in research, it has growing relevance for production use due to its performance and flexibility. Its suitability for inference on embedded systems is still an active area of exploration, especially in comparison to frameworks like TensorRT or TVM that offer dedicated deployment optimizations.

4. Methodology

This study conducts a systematic benchmarking of five widely adopted DL inference frameworks, PyTorch, ONNX Runtime, TensorRT, Apache TVM and JAX, on the NVIDIA Jetson AGX Orin platform. To ensure consistency across comparisons, identical models, datasets, preprocessing, and evaluation protocols were used. The experiments were carefully designed to highlight how each framework scales across a range of batch sizes and responds to hardware-aware optimizations relevant to edge computing.

4.1. Dataset and Preprocessing

All evaluations were performed using the ImageNet ILSVRC 2012 validation dataset, which includes 50,000 high-resolution natural images classified into 1000 categories. Preprocessing involved resizing images to 256 × 256 pixels, followed by center cropping to 224 × 224 pixels. Standard normalization using mean values of [0.485, 0.456, 0.406] and standard deviations of [0.229, 0.224, 0.225] was applied to align with the preprocessing pipeline of ImageNet-trained models [27]. These preprocessing steps were consistently implemented across all five frameworks to ensure input uniformity.

4.2. Model Selection and Conversion

Six pretrained models were selected to represent a diverse range of neural network architectures and computational complexities. These include ResNet-152 [28], MobileNetV2 [29], SqueezeNet [8], EfficientNet-B0 [30], VGG16 [27], Swin Transformer [31], and YOLOv5s [32]. All models were sourced from the PyTorch model zoo and initially implemented in PyTorch. For interoperability and benchmarking, models were exported to ONNX format with dynamic axes enabled, enabling execution across various inference frameworks. ONNX Runtime was evaluated using the GPU execution provider exclusively. For TensorRT, conversion was performed via ONNX using the PyTorch-to-TensorRT pipeline. Meanwhile, both JAX and TVM leveraged PyTorch models as backends for their respective optimization workflows, rather than relying on ONNX conversion. This setup ensured consistent model definitions across frameworks while allowing evaluation of native optimization strategies.

To ensure fair performance comparison, we evaluated all models across a range of batch sizes: 2, 4, 8, 16, 32, 64, and 128. However, for JAX, inference was only conducted for batch sizes 2, 4, and 8, as larger batch sizes exceeded the available memory and could not be executed. Additionally, to stabilize timing measurements, 50 warmup runs were performed prior to recording inference times for all frameworks and batch sizes. Tables 1 and 2 indicate the details of selected models and optimizations we applied for each framework, respectively.

Table 1. Details of selected models.

Model	Params (M)	Representativeness
ResNet-152 [28]	60.2	Deep, high-capacity CNN used as a benchmark for classification tasks.
MobileNetV2 [29]	3.4	Lightweight architecture optimized for mobile and edge devices.
SqueezeNet [8]	1.2	Extremely compact model suitable for highly resource-constrained environments.
EfficientNet-B0 [30]	5.3	Balances accuracy and efficiency; commonly used in edge scenarios.
VGG16 [27]	138	High-capacity model with historical significance; included for comparative completeness.
Swin Transformer [31]	29	Vision transformer model; representative of recent architectural trends.
YOLOv5s [32]	7.2	Real-time object detector widely adopted in embedded and edge AI applications.

Table 2. Optimization techniques applied for each framework.

Framework	Precision Used	Graph Optimization	Notes
PyTorch	FP16	cuDNN backend	Native inference with torch.autocast and AMP
ONNX Runtime	FP16	Graph fusion (default)	Used ExecutionProvider with graph optimizations enabled
TensorRT	FP16	Layer fusion, kernel tuning	Converted from ONNX using FP16 mode in builder config
TVM	FP16	AutoTVM tuning	Used Relax IR with tuning logs; failed for SqueezeNet
JAX	FP16	XLA JIT compilation	Enabled 16-bit precision using jax.numpy and XLA flags

4.3. PyTorch Inference

PyTorch served as the baseline framework, using models in evaluation mode with CUDA acceleration enabled. Input images were processed in batches with sizes ranging from 2 to 128 to evaluate scalability. To enhance performance, the models were converted to TorchScript using tracing, which compiles the computation graph and removes Python-level overhead. Additionally, cuDNN auto-tuner benchmarking was enabled to select the fastest convolution algorithms dynamically during runtime [11]. The inference loop was GPU-synchronized using CUDA events to ensure accurate latency measurement. Memory usage was monitored through PyTorch's built-in memory tracker, and accuracy was computed using top-k classification metrics.

4.4. ONNX Runtime Inference

Inference using ONNX Runtime was carried out using CUDAExecutionProvider backends. The ONNX format enabled the framework to apply a series of graph-level optimizations automatically. These include constant folding, operator fusion, and elimination of redundant initializers, performed via the *ORT_ENABLE_ALL* optimization level [20]. Batch sizes from 2 to 128 were evaluated for each model. NumPy was used to prepare and feed data into the ONNX Runtime sessions, and predictions were extracted from model outputs for accuracy computation. In all configurations, ONNX Runtime demonstrated competitive portability and flexibility, making it well-suited for multi-platform deployment scenarios.

4.5. TensorRT Inference

TensorRT is NVIDIA's high-performance DL inference SDK designed to optimize trained models for deployment [12]. The ONNX models were imported into TensorRT and converted into serialized engines. FP16 precision was enabled to take advantage of the Jetson AGX Orin's native hardware acceleration for half-precision operations, which reduces memory footprint and increases throughput. During engine building, we enabled advanced optimization techniques such as layer and tensor fusion, kernel auto-tuning, dynamic shape support, and memory reuse. The engines were deployed with input buffers allocated in GPU memory and executed using asynchronous inference calls managed through CUDA streams. TensorRT's ability to generate highly optimized kernels for specific batch sizes made it one of the most performant frameworks, especially under larger batch configurations.

4.6. Apache TVM Inference

Apache TVM is a DL compiler stack designed to optimize and deploy models across heterogeneous hardware platforms [14]. In our workflow, PyTorch models were directly imported into TVM via its PyTorch frontend, bypassing the need for ONNX conversion. The models were translated into TVM's Relax intermediate representation (IR) and compiled for the CUDA-enabled GPU on the NVIDIA Jetson Orin platform. TVM applied both graph-level and tensor-level optimizations, including operator fusion, memory layout transformation, and loop unrolling. AutoTVM was used to tune kernel schedules using an empirical cost model. For each tested batch size (ranging from 2 to 128), TVM recompiled the model to adapt memory allocation and computational parallelism accordingly. The compiled modules were executed using TVM's graph executor. TVM's low-level control over compilation and its backend-specific autotuning capabilities made it particularly effective for edge inference optimization on Jetson devices.

4.7. JAX Inference

JAX is a high-performance numerical computing library that compiles NumPy-compatible Python code using XLA for execution on accelerators [15,33]. In our setup, model computation graphs were restructured using functional transformations and compiled using JAX's `jit`. JAX supported mixed-precision operations using bfloat16 and enabled fusion of elementwise operations during XLA compilation. To reduce startup latency, warm-up iterations were run before recording performance measurements. The inference function was compiled just-in-time and optimized for both compute and memory efficiency. JAX's composability and XLA's backend-specific optimizations allowed it to achieve high performance with minimal framework-specific tuning.

4.8. Inference Performance Measuring

4.8.1. Model Accuracy

We evaluated the classification accuracy of models deployed across all frameworks using the ImageNet ILSVRC 2012 validation dataset, which contains 50,000 labeled images spanning 1000 classes [34]. For each model and batch size, Top-1 and Top-5 classification accuracy metrics were computed.

Top-1 accuracy quantifies the percentage of test samples for which the model's highest probability prediction matches the ground truth label. Top-5 accuracy considers a prediction correct if the ground truth label is among the model's five most probable predictions. These metrics were computed using framework-native methods such as `torch.topk` in PyTorch, `np.argsort` for ONNX Runtime, and `jax.lax.top_k` for JAX.

4.8.2. Inference Time Measurement

Inference time was measured as the average execution time of a forward pass through the model, excluding preprocessing and model loading phases. To account for runtime optimization and caching effects, we included a warm-up phase of 50 inferences prior to measurement, as supported by prior literature [22]. Timing was captured using high-resolution timestamps (e.g., `time.time()` or CUDA events) immediately before and after inference execution.

To ensure fair comparison, latency measurements were synchronized with device-specific barriers, such as CUDA stream synchronizations in PyTorch and TensorRT, and internal session synchronizations in ONNX Runtime and TVM.

4.8.3. Throughput Calculation

Throughput was defined as the number of processed inputs per second and computed as the ratio of the batch size to the average inference time per batch. This approach follows previous guidelines from work like Xu et al. [35]. We experimented with batch sizes ranging from 2 to 128, adjusting based on the available system memory capacity.

Frameworks like TensorRT and Torch-TensorRT support engine construction optimized for fixed batch sizes, which enables higher throughput. Conversely, PyTorch, ONNX Runtime, and JAX support dynamic batching and runtime kernel selection. These characteristics influence how each framework scales throughput with batch size.

To prevent memory errors during ONNX export or inference, especially at larger batch sizes, we designated the batch dimension as a dynamic axis during model conversion from PyTorch to ONNX. This strategy mitigated known memory issues linked to internal tensor duplication [36].

4.8.4. System Memory and Power Monitoring

To collect runtime telemetry on power and memory usage during inference, we employed a parallelized monitoring approach integrated with the Jetson AGX Orin's onboard sensors and external libraries.

Power draw was recorded using the `jtop` interface from the Jetson Stats toolkit. A dedicated background thread was spawned to maintain a persistent connection with the `jtop` daemon. This thread periodically queried telemetry fields such as `POM_5V_IN`, `VDD_IN`, and `VDD_SOC` from the system power dictionary, selecting the most stable reading available.

System memory usage was tracked concurrently within the same monitoring thread by accessing RAM metrics from `jtop`'s memory module. Specifically, the `used` field under the `RAM` key was queried and converted to megabytes. Given the unified memory architecture of Jetson AGX Orin, this value reflects the total DRAM consumption attributable to the inference process, encompassing both CPU and GPU allocations.

All telemetry sampling was offloaded to an isolated thread, ensuring non-blocking behavior with respect to the main inference pipeline. This allowed real-time monitoring without introducing synchronization delays or overhead to the model execution flow. All data were sampled at a frequency of 10 Hz (i.e., every 100 ms), covering the entire active inference interval for each configuration. The collected values were averaged post-execution to compute the mean during inference run.

4.9. Hardware Specifications

NVIDIA has developed a line of embedded computing devices called NVIDIA® JetsonTM, which are intended for use in machine learning and artificial intelligence applications. These are feature-rich, low-power, and compact systems that are capable of executing sophisticated DL models instantly. We do our research using the NVIDIA Jetson AGX Orin 64GB, the newest Jetson device manufactured by NVIDIA Corp., Santa Clara, CA, USA. It has a 12-core NVIDIA Carmel ARMv8.2 CPU, a 384-core NVIDIA Volta GPU, and a 32-core NVIDIA Deep Learning Accelerator (DLA) [37]. It is built on the NVIDIA Ampere architecture.

4.10. Software Specifications

All software tools used in this study were selected to be compatible with the NVIDIA Jetson AGX Orin platform, which requires strict alignment between versions of system libraries and acceleration frameworks. For instance, TensorRT 8.6.2.3 requires CUDA 12.2 and is tightly coupled with specific versions of cuDNN and ONNX parsers. Ensuring compatibility across these components is essential for stable deployment and accurate benchmarking.

The experiments were conducted using JetPack 6.0, which includes CUDA 12.2 and cuDNN 8.9.4 as part of its software stack. Python 3.10 was the base interpreter for all benchmarking scripts. Each DL inference framework used in this study was installed in versions that are stable and optimized for the Jetson Orin platform. Table 3 summarizes the key software tools and library versions used throughout this work.

Table 3. Software tools and library versions used.

JetPack	CUDA	ONNX	ONNX Runtime	PyTorch	TensorRT	Torch-TensorRT	TensorFlow	JAX	TVM
6.0	12.2	1.17.0	1.17.1	2.3.0	8.6.2.3	1.4.0	2.15.0	0.4.28	0.21.0

5. Experimental Results

5.1. Inference Output Validation

The Top-1 accuracy results are presented in Table 4, while the corresponding Top-5 accuracy scores are shown in Table 5. PyTorch, which served as the baseline framework, demonstrated consistent performance across all models. Its close integration with the original model weights ensured minimal deviation, and it provides a reference against which other backends were evaluated.

ONNX Runtime closely matched PyTorch's performance across most models, showing variations within a 1–2% range. This slight deviation is expected due to backend-specific optimizations such as operator fusion and constant folding. In certain models like MobileNet and VGG16, ONNX Runtime slightly outperformed PyTorch in Top-1 accuracy, likely benefiting from optimized execution paths.

TensorRT achieved the highest Top-1 accuracy for ResNet152 and EfficientNet, reaching 76.64% and 74.72%, respectively. These results, as highlighted in Table 4, affirm the effectiveness of its layer fusion and precision calibration strategies. However, small degradations were noted for models like SqueezeNet and Swin Transformer, indicating potential sensitivity of lightweight and transformer architectures to aggressive quantization.

Apache TVM exhibited robust accuracy in most cases and equaled or surpassed other frameworks for models like VGG16 and Swin Transformer. As seen in Table 5, TVM reported a Top-5 accuracy of 94.60% for Swin Transformer, the highest among all evaluated frameworks. The absence of results for SqueezeNet suggests potential compatibility issues during model import or relax graph lowering.

JAX displayed notably high accuracy for several models, including EfficientNet and ResNet152, as observed in both Tables 4 and 5. These elevated figures may stem from architectural differences in how JAX compiles and executes functions via XLA, as well as discrepancies in model conversion or preprocessing pipelines. These results warrant further scrutiny and may benefit from reproducibility analysis with shared checkpoints.

Table 4. Top-1 Accuracy (%).

Inference Framework	ResNet152	MobileNet	SqueezeNet	EfficientNet	VGG16	Swin Transformer	YOLOv5s
PyTorch	75.30	69.70	56.70	74.00	68.00	77.80	69.70
ONNX Runtime	72.00	70.50	55.00	73.50	69.06	76.50	67.50
TensorRT	76.64	70.60	56.06	74.72	68.06	75.68	69.80
TVM	74.26	71.20	NA	74.40	70.37	77.80	69.60
JAX	72.00	70.80	55.30	74.00	68.54	77.00	67.00

Table 5. Top-5 Accuracy (%).

Inference Framework	ResNet152	MobileNet	SqueezeNet	EfficientNet	VGG16	Swin Transformer	YOLOv5s
PyTorch	94.13	90.30	80.63	93.10	88.00	94.90	89.60
ONNX Runtime	94.50	91.00	80.57	93.00	90.87	93.50	90.50
TensorRT	93.38	89.64	78.98	91.88	88.24	92.86	89.18
TVM	93.80	90.40	NA	92.40	89.80	94.60	89.80
JAX	93.60	90.05	79.07	92.50	89.60	93.00	89.70

Observation 1. All frameworks maintained high classification accuracy, with only minor deviations from the PyTorch baseline. Overall, accuracy remained robust across frameworks, validating their use for deployment without significant accuracy loss.

5.2. Inference Time

We evaluated the average inference time per input sample across six representative DL models: ResNet152, MobileNet, SqueezeNet, EfficientNet, VGG16, and Swin Transformer.

The results for each framework, PyTorch, ONNX Runtime, TensorRT, TVM, and JAX, are reported in Table 6 and visualized in Figure 1 using a logarithmic scale on the vertical axis for improved readability across orders of magnitude.

TensorRT emerged as the most efficient backend, consistently producing the lowest inference times across all models. Its performance benefits from several tightly integrated optimizations such as FP16 precision support, kernel auto-tuning, and layer fusion during static engine construction. For instance, it achieved inference times of approximately 0.66 ms for SqueezeNet and 2.28 ms for ResNet152, indicating its capability to scale well across both lightweight and heavy architectures.

PyTorch followed as the second-fastest framework, delivering moderate-to-low latency across all tested models. Its backend leverages highly optimized CUDA and cuDNN libraries but lacks static graph-level optimization. Nevertheless, inference times were within a practical range for edge deployment, with MobileNet and SqueezeNet completing forward passes in under 5 ms per sample.

In contrast, ONNX Runtime showed significantly higher latency, especially for deeper models. ResNet152 and VGG16 required over 280 ms and 158 ms, respectively, more than an order of magnitude slower than TensorRT. These results suggest that ONNX Runtime's execution providers may not be fully optimized for the Jetson Orin platform, particularly in terms of kernel fusion and hardware-specific execution planning.

TVM's performance varied by model. It achieved good latency for transformer-based and larger convolutional models such as Swin Transformer and VGG16, but performed less efficiently on MobileNet and could not successfully compile SqueezeNet. This variation reflects both the power and current limitations of AutoTVM tuning and Relax compilation on Jetson-class embedded hardware.

JAX offered moderate latency values, consistently trailing behind TensorRT and PyTorch. While its XLA-based JIT compilation allows for graph-level optimization, the general-purpose nature of the compiler and lack of hardware-specific tuning may limit its performance on edge devices. Nonetheless, it performed predictably across all models, without outliers or instability.

Table 6. Average Inference Time per Sample (ms).

Inference Framework	ResNet152	MobileNet	SqueezeNet	EfficientNet	VGG16	Swin Transformer	YOLOv5s
PyTorch	9.239	4.201	1.749	4.971	1.713	7.267	2.300
ONNX Runtime	285.534	54.402	21.270	89.842	158.146	19.813	26.287
TensorRT	2.282	1.136	0.662	1.516	2.195	3.949	0.882
TVM	7.429	9.529	NA	12.695	8.908	5.275	4.662
JAX	29.096	11.235	8.255	15.182	19.326	22.268	11.686

Observation 2. TensorRT consistently delivered the fastest inference times across all models, demonstrating its strong suitability for real-time applications on edge devices. PyTorch followed closely, offering competitive latency with minimal optimization effort. ONNX Runtime showed significantly higher latency, particularly for larger models, limiting its practicality for time-sensitive tasks. TVM and JAX exhibited variable performance, with TVM excelling on some models but struggling with others, and JAX offering consistent yet slower inference. Overall, TensorRT clearly leads in execution efficiency on the Jetson AGX Orin platform.

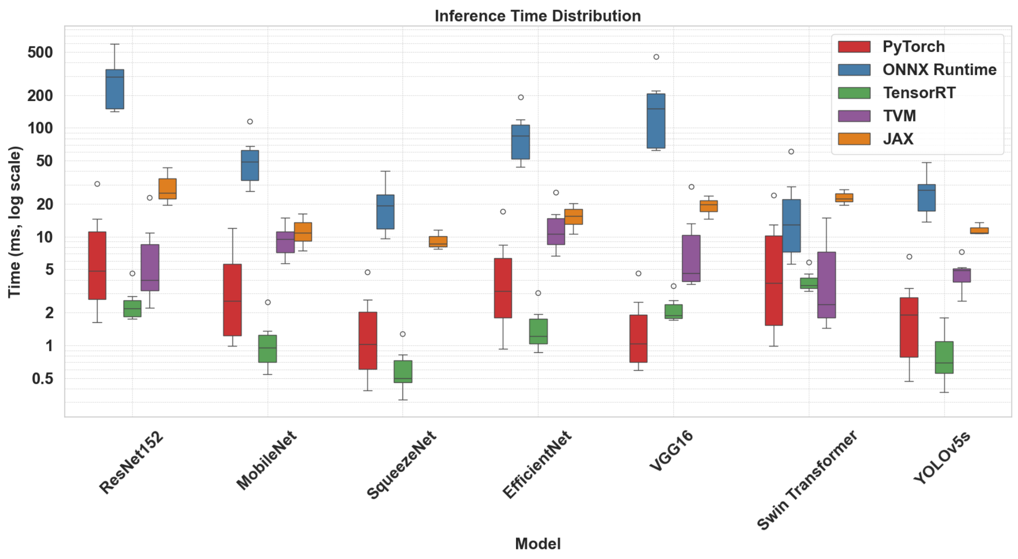

Figure 1. Inference Time Distribution (y axis in log scale).

5.3. Inference Throughput

In our evaluation, we calculated the maximum achievable throughput for each model across all frameworks by incrementally increasing the batch size until the highest sustainable rate was observed without exceeding system memory limits. The results are summarized in Table 7 and visualized in Figure 2. Additionally, Figure 3 presents a detailed view of throughput variation with batch size for MobileNet as a representative lightweight model.

TensorRT achieved the highest throughput for nearly all models, most notably reaching 3197.10 samples/s on SqueezeNet and 1382.46 samples/s on MobileNet. These results reflect TensorRT's ability to exploit static graph optimizations, layer fusion, and efficient kernel scheduling. Its performance scaled well with increasing batch sizes, owing to its use of precompiled, batch-optimized inference engines.

PyTorch also demonstrated impressive throughput, particularly on MobileNet, EfficientNet, and VGG16. Although it does not statically optimize execution graphs, PyTorch's dynamic computation engine and cuDNN-based backend allow it to benefit from increased batch size. As shown in Figure 3, PyTorch maintained high throughput even at large batch sizes, suggesting efficient GPU memory usage and parallelism.

ONNX Runtime underperformed relative to other frameworks. Its throughput peaked at only 181.46 samples/s for Swin Transformer and was significantly lower for deeper models such as ResNet152 and VGG16. This can be attributed to the lack of kernel-level fusion and less aggressive scheduling on Jetson's hardware. Despite being portable and extensible, ONNX Runtime appears limited in raw throughput performance in embedded settings.

TVM achieved moderate throughput across most models and notably surpassed PyTorch and TensorRT for Swin Transformer, reaching 874.48 samples/s. This suggests that AutoTVM tuning was particularly effective for transformer-based architectures, possibly due to efficient relay-to-CUDA scheduling. However, it failed to compile SqueezeNet, limiting its universality.

JAX reported consistent but modest throughput, with best performance on MobileNet (126.02 samples/s) and SqueezeNet (132.34 samples/s). These values suggest that JAX's general-purpose XLA compilation and runtime optimizations provide portability and correctness but fall short in low-level throughput scaling, particularly under large batch sizes.

Table 7. Maximum Acheived Throughput (samples/s).

Inference Framework	ResNet152	MobileNet	SqueezeNet	EfficientNet	VGG16	Swin Transformer	YOLOv5s
PyTorch	931.81	1058.85	1887.53	1202.90	1859.26	1250.40	2124.81
ONNX Runtime	7.45	37.63	106.02	23.50	16.34	181.46	72.84
TensorRT	651.67	1382.46	3197.10	1137.63	582.85	317.52	2691.02
TVM	456.22	171.28	NA	140.58	486.07	874.48	388.04
JAX	50.33	126.02	132.34	95.82	73.34	49.46	93.68

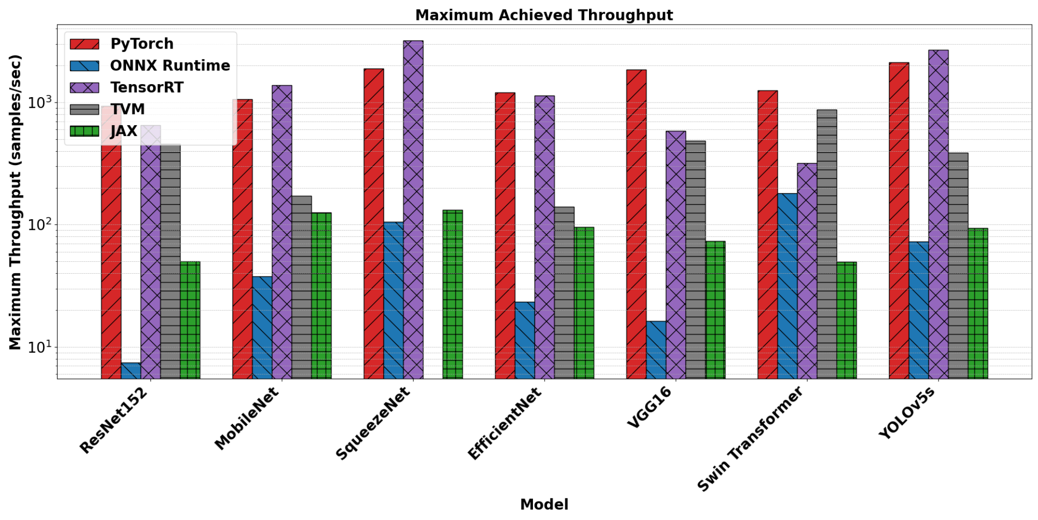

Figure 2. Maximum Achieved Throughput for All Models (y axis in log scale).

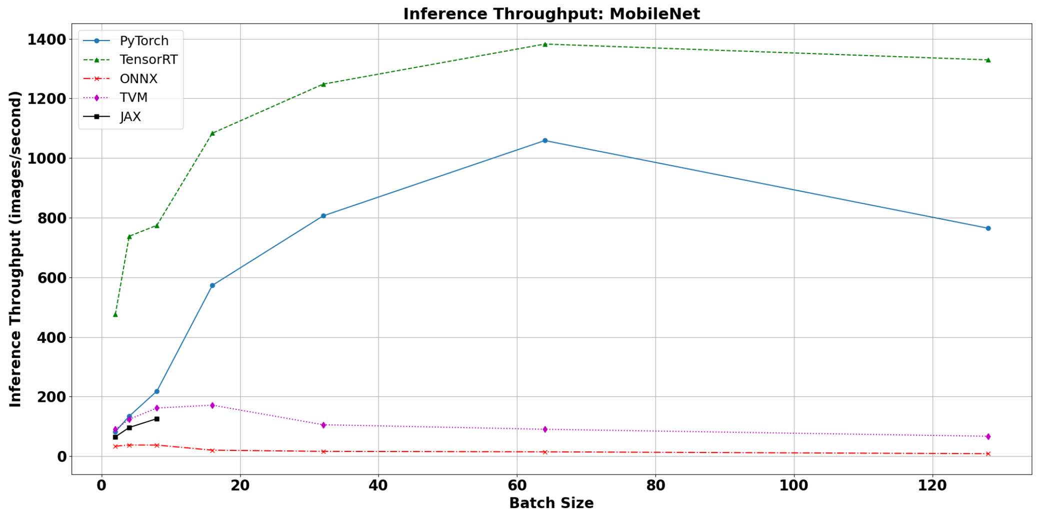

Figure 3. Inference throughput for Mobilenet with varying batch size.

Observation 3. TensorRT delivers the highest throughput across most models, driven by its batch-tuned engine and hardware-aware optimizations. PyTorch follows closely, demonstrating excellent scalability with increasing batch sizes. ONNX Runtime lags significantly in throughput, while TVM shows strength in transformer models. JAX maintains stability but does not scale as effectively. These results highlight TensorRT and PyTorch as top candidates for high-throughput deployment on embedded platforms.

5.4. System Memory Utilization

We recorded the average system memory usage (in GB) during inference across all models and frameworks, as shown in Table 8. Additionally, Figure 4 illustrates how memory usage scales with batch size for MobileNet.

TVM consistently demonstrated the most memory-efficient behavior, consuming the least memory across all test cases. For example, it maintained MobileNet inference within approximately 10.3 GB regardless of batch size, indicating effective memory planning and lightweight runtime overhead. This efficiency likely stems from TVM's static memory allocation strategies and custom graph compilation that minimizes redundant allocations.

PyTorch and TensorRT exhibited moderate and closely aligned memory footprints, with average usage between 14–16 GB across models. Their dynamic memory management, coupled with reliance on cuDNN and CUDA memory allocators, results in predictable scaling with batch size. Notably, Figure 4 shows smooth and steady increases in memory consumption for both frameworks, reflecting stable buffer reuse and kernel launch behavior.

ONNX Runtime consumed more memory than both PyTorch and TensorRT. For MobileNet, its usage hovered around 17.7 GB and showed minimal fluctuation with batch size. This higher footprint may arise from internal graph transformations or redundant buffer allocations during execution.

JAX showed significantly higher memory usage across all models, averaging nearly 29 GB. This result is consistent with its functional programming model, where immutable tensor structures and just-in-time compilation introduce memory overhead. As illustrated in Figure 4, JAX's usage remains static and elevated, indicating that it does not optimize memory use per batch dynamically.

Table 8. Average Memory Usage (GB).

Inference Framework	ResNet152	MobileNet	SqueezeNet	EfficientNet	VGG16	Swin Transformer	YOLOv5s
PyTorch	12.039	14.152	14.968	15.584	16.805	18.484	16.180
ONNX Runtime	15.420	17.693	18.164	18.745	21.134	19.853	18.537
TensorRT	12.279	14.120	14.843	15.746	17.368	17.507	14.554
TVM	7.773	10.287	NA	10.926	13.717	15.275	14.926
JAX	28.974	28.799	28.703	28.663	28.941	29.193	29.634

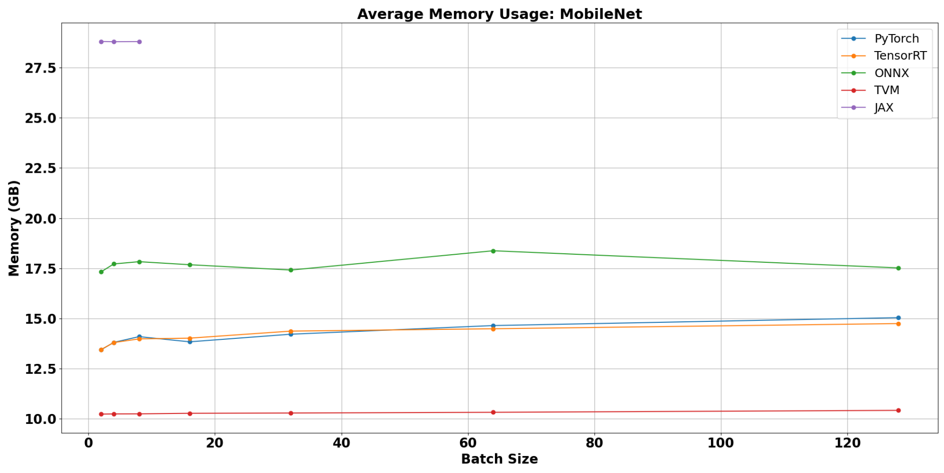

Figure 4. System Memory Usage for Mobilenet.

Observation 4. TVM offers the most efficient memory usage, making it suitable for memory-constrained environments. PyTorch and TensorRT strike a balance between performance and moderate memory overhead. ONNX Runtime uses more memory than expected, while JAX consistently consumes the most, limiting its scalability. These results underscore the importance of framework-level memory handling in real-time edge deployment.

5.5. System Power Consumption

We measured the average system power consumption during inference, using the onboard telemetry sensors of the NVIDIA Jetson AGX Orin. The results, averaged across batch sizes and inference cycles, are presented in Table 9.

TensorRT recorded the highest power draw among all frameworks, particularly when running larger models such as ResNet152 (28.30 W), VGG16 (29.85 W), and Swin Transformer (27.43 W). This elevated consumption aligns with its high throughput and low latency performance, indicating that TensorRT aggressively utilizes hardware resources, including GPU cores and memory bandwidth, to maximize inference speed. Its power profile suggests a performance-at-all-costs design philosophy, suitable for scenarios where latency is paramount and power is not a limiting factor.

PyTorch also exhibited relatively high power consumption, especially on deep CNN models like VGG16 and ResNet152, consuming 23.44 W and 21.83 W, respectively. These values reflect its use of dynamic graph execution and frequent GPU kernel launches, which, while efficient, may not fully optimize power-aware scheduling compared to compiled runtimes.

ONNX Runtime and JAX demonstrated the lowest average power consumption across most models, remaining below 15 W even for large networks. For example, ONNX Runtime drew only 14.17 W on ResNet152 and 13.96 W on EfficientNet, indicating modest GPU engagement. However, these energy savings come at the cost of higher inference latency and lower throughput, as shown in previous sections. JAX followed a similar trend, with the lowest power draw on MobileNet (11.27 W) and SqueezeNet (10.66 W), which reflects limited kernel-level parallelism despite being functionally correct and stable.

TVM's power footprint varied across models. It was competitive with PyTorch on large models, showing 23.62 W on VGG16 and 21.86 W on ResNet152, but consumed less on smaller models. This intermediate behavior suggests that TVM's auto-tuned compilation can yield efficient execution patterns but may still require hardware-specific power optimizations to match the balance achieved by TensorRT.

Table 9. Average Power Consumption (W).

Inference Framework	ResNet152	MobileNet	SqueezeNet	EfficientNet	VGG16	Swin Transformer	YOLOv5s
PyTorch	21.832	14.442	14.306	17.096	23.440	18.802	14.321
ONNX Runtime	14.166	12.459	13.312	13.962	14.213	13.783	13.176
TensorRT	28.299	13.062	11.789	17.557	29.851	27.430	12.608
TVM	21.858	13.766	NA	15.156	23.620	16.045	12.250
JAX	15.076	11.269	10.658	12.229	15.736	15.331	10.522

Observation 5. TensorRT delivers the highest inference performance with the highest power consumption, especially for large models. ONNX Runtime and JAX are the most power-efficient but also the slowest. PyTorch and TVM offer a middle ground, with moderately high power draw and good speed. These results highlight the trade-off between energy efficiency and computational performance in edge deployment scenarios.

6. Discussion

This study provides a comprehensive evaluation of modern DL inference frameworks. Our analysis spans a wide range of performance indicators, including inference accuracy, inference time, throughput, system memory utilization, and power consumption. The diversity of the selected models, encompassing both convolutional and transformer-based architectures, allows us to capture framework behavior under varying computational and memory demands.

Our results clearly illustrate that no single framework dominates across all performance dimensions. Each exhibits strengths aligned with its design philosophy, exposing important tradeoffs for deployment engineers and system architects.

TensorRT stands out in raw performance. It achieves the lowest inference latency and highest throughput across most models, demonstrating its effectiveness as a production-grade inference engine optimized for NVIDIA hardware. Its static engine construction, support for reduced precision computation (e.g., FP16), and advanced kernel fusion enable highly efficient GPU utilization [12]. However, this performance comes at the cost of power efficiency: TensorRT exhibits the highest system power draw among all frameworks. For edge devices with tight thermal envelopes or power budgets, this may necessitate tradeoffs between speed and energy efficiency.

PyTorch achieves a balance between performance and usability. Although not as fast as TensorRT, it maintains relatively low latency and strong throughput, especially when batch sizes are scaled appropriately. This performance is attributable to its mature cuDNN backend and dynamic computation graph, which allow for flexible model execution with minimal tuning. PyTorch also integrates well with deployment tools like TorchScript and Torch TensorRT, making it an accessible and adaptable solution for both development and deployment. Nevertheless, the absence of deep hardware-aware graph compilation restricts PyTorch from reaching the peak efficiency seen in statically compiled runtimes.

ONNX Runtime, while conceived as a highly portable and interoperable inference backend, significantly lags behind TensorRT and PyTorch in both latency and throughput. This performance gap becomes especially evident in deeper networks like ResNet152 and VGG16. ONNX Runtime's higher system memory usage further complicates its deployment on memory-constrained embedded devices. Despite these drawbacks, ONNX's value lies in its role as a standardized model interchange format [38], supporting cross-framework workflows and enabling interoperability across diverse hardware ecosystems. Improvements to ONNX Runtime's backend optimizations, particularly its TensorRT Execution Provider, may help bridge this performance gap in future versions.

Apache TVM presents an interesting middle ground. It demonstrates excellent system memory efficiency, likely due to its low-level code generation and memory planning strategies [14]. TVM also performs competitively in throughput, especially for transformer models like Swin Transformer, where its tuning strategies appear well matched to GPU parallelism. However, TVM's instability, evidenced by failures to compile SqueezeNet and erratic performance for certain models, exposes limitations in operator coverage and frontend robustness. This restricts its general purpose usability despite its high theoretical efficiency.

JAX, with its functional programming paradigm and XLA-backed just-in-time compilation, offers a unique inference model that emphasizes composability and reproducibility. It consistently achieved the lowest system power consumption, making it appealing for energy sensitive deployments. Yet, its high memory usage and modest throughput point to limitations in runtime execution planning and memory reuse, perhaps a result of its general purpose compiler not being fully tuned for real time inference on embedded platforms. These characteristics suggest that JAX may be more suitable for experimental settings or academic research rather than production edge inference.

While our empirical results highlight distinct performance characteristics across inference frameworks, a theoretical analysis of their internal mechanisms offers further clarity into these outcomes. TensorRT's superior latency and throughput can be attributed to its use of aggressive optimization strategies, such as precision quantization (e.g., FP16), kernel auto-tuning, and operation fusion. These transformations reduce memory bandwidth bottlenecks and improve parallelism by lowering compute granularity and minimizing runtime overheads [12]. Unlike dynamic frameworks, TensorRT performs static compilation of ONNX graphs into highly optimized CUDA kernels, thereby bypassing Python-level execution constraints and enabling deterministic execution paths. In contrast, JAX, while demonstrating low power usage, exhibited the highest memory consumption. This can be theoretically understood through the lens of its design philosophy: JAX leverages XLA for ahead-of-time compilation, and its functional programming model emphasizes immutability. These properties, while beneficial for correctness and reproducibility, lead to extensive memory allocation, as intermediate tensors cannot be reused in-place [15,33]. This memory-intensive behavior may hinder its utility in constrained embedded environments. Apache TVM, known for fine-grained operator tuning and low-level IR transformations [14], showed commendable memory efficiency and throughput. However, its failure to compile certain models like SqueezeNet reveals architectural limitations in operator coverage, particularly within the evolving Relax IR. These limitations restrict TVM's out-of-the-box compatibility with certain ONNX-exported graphs, especially those containing custom or fused operators unsupported by the current compiler stack. Importantly, these framework behaviors are sensitive to version volatility. Both TVM and ONNX Runtime undergo rapid development cycles, which may introduce breaking changes or varying support levels for optimization passes and hardware accelerators. As such, reproducibility across future deployments may be affected unless version constraints are strictly maintained and documented. Including structured documentation of such version-dependent behaviors, along with a comparative summary of framework-specific constraints and potential mitigation strategies, would enhance the practicality of benchmarking results for deployment engineers and system designers. Table 10 summarizes the limitations of frameworks and probable mitigation strategies.

A key strength of this work lies in the reproducible and controlled benchmarking methodology. By standardizing the evaluation environment, dataset (ImageNet validation set), preprocessing pipeline, hardware platform, and metric definitions, we ensure fair and interpretable comparisons across frameworks. The use of six models from diverse architecture families strengthens generalizability, highlighting how different computational patterns (e.g., dense convolutions vs. self-attention) affect framework performance.

Furthermore, we go beyond traditional latency and accuracy measurements by incorporating throughput, system memory consumption, and power usage, which are often neglected in prior works [22,35]. These metrics are crucial for real world deployment on constrained devices, providing a holistic view of each framework's operational cost and scalability.

Despite its strengths, the study has several limitations. First, it focuses exclusively on inference; training performance, while less relevant for deployment, is important in certain edge learning scenarios (e.g., federated learning or continual learning). Second, the analysis is restricted to the Jetson AGX Orin platform. While this device is representative of modern edge accelerators, performance characteristics may differ on other embedded hardware such as Google Coral, Intel Movidius, or AMD-based systems. Extending the study across multiple hardware targets would provide broader insights into cross-platform optimization.

Additionally, the framework versions and software stacks used in this study reflect a snapshot in time. These ecosystems are evolving rapidly. For example, ONNX Runtime and TVM continue to release performance enhancements that may alter current results. Keeping benchmarks up to date is essential for ensuring relevance as frameworks improve and hardware capabilities expand. The rapid development of frameworks like ONNX Runtime and Apache TVM leads to frequent updates that can affect performance, compatibility, and operator support [14]. These changes may hinder reproducibility across versions, especially in embedded systems where software stacks are tightly integrated.

To address this, documenting software versions and using containerized environments is recommended for consistent deployment.

Finally, certain runtime behaviors, such as kernel caching, compiler warm up, and background telemetry, introduce variability that may affect repeatability at a fine grained level. While our methodology includes warm up runs and averaged results, further statistical rigor (e.g., confidence intervals or variance analysis) could enhance the precision of reported metrics.

The results of this study provide actionable guidance for edge AI deployment. Developers seeking maximum speed, such as for autonomous navigation or high frame rate vision tasks, will benefit from TensorRT, provided power constraints are relaxed. For general purpose deployments, PyTorch offers a productive tradeoff between performance and portability. Memory or energy-constrained use cases may prefer TVM or JAX, depending on stability and hardware alignment. ONNX Runtime, while underperforming in this study, remains valuable for its ecosystem interoperability and may serve as a reliable fallback when framework flexibility is required.

Table 10. Framework-Specific Limitations, Causes, and Mitigation Strategies.

Framework	Limitation	Likely Cause	Mitigation Strategy
PyTorch	Moderate power and memory usage; lacks static graph-level optimization	Dynamic computation graph; no ahead-of-time compilation	Using TorchScript tracing; integrating Torch-TensorRT; optimizing memory allocators
ONNX Runtime	High latency and memory usage (e.g., 21.13 GB for VGG16)	Limited optimization for Jetson hardware; inefficient graph execution	Enabling TensorRT Execution Provider; applying manual optimization passes; avoiding very deep models
TensorRT	High power consumption (approximately 30 W for large models)	Aggressive GPU usage, kernel fusion, and FP16 throughput focus	Using INT8 quantization; limiting batch size; enforcing runtime power constraints
TVM	Compilation failure for SqueezeNet	Incomplete operator coverage or Relax IR limitations	Extending operator support; reverting to Relay IR; simplifying model architecture
JAX	Excessive memory usage (around 29 GB) across models	Functional paradigm with immutable tensors and XLA compilation overhead	Using smaller batch sizes; profiling with XLA tools; manually reusing tensors where feasible

7. Conclusions

In this study, we conducted a detailed comparative analysis of five widely used DL inference frameworks: PyTorch, ONNX Runtime, TensorRT, Apache TVM, and JAX on the NVIDIA Jetson AGX Orin platform. By evaluating diverse pretrained models on the ImageNet validation dataset, we assessed each framework across multiple dimensions, including inference accuracy, latency, throughput, system memory usage, and power consumption.

Our results show that each framework offers unique strengths depending on the deployment context. TensorRT delivered the fastest inference and highest throughput, confirming its suitability for high performance applications where speed is critical. However, this performance came with increased power consumption. PyTorch emerged as a strong general purpose framework, balancing usability with efficient runtime execution. ONNX Runtime, while versatile in terms of model portability, showed relatively lower performance, suggesting room for further optimization. TVM demonstrated impressive memory efficiency and competitive throughput in specific scenarios, although it faced stability issues with certain models. JAX stood out for its low power consumption and functional design but lagged in memory efficiency and raw performance.

These findings emphasize the importance of selecting inference frameworks based not only on speed or accuracy but also on practical deployment factors like energy use, memory footprint, and hardware compatibility. Our work provides developers and researchers with concrete benchmarks and insights to guide framework selection for embedded AI applications.

Future work will extend this study to include additional hardware platforms, updated software stacks, and training performance metrics, offering a broader perspective on DL deployment in resource constrained environments.

Author Contributions: All authors contributed significantly to the work. All authors have read and agreed to the published version of the manuscript.

Funding: This work is supported in part by NSF grants 2104181 and 2442078.

Data Availability Statement: All experiments were conducted using publicly availble ImageNet validation dataset, which can be accessed at https://www.image-net.org.

Conflicts of Interest: The authors declare that there are no conflicts of interest regarding the publication of this paper.

References

1. Image Classification on ImageNet. Available online: https://paperswithcode.com/sota/image-classification-on-imagenet (accessed on 20 July 2025).
2. Devlin, J.; Chang, M.W.; Lee, K.; Toutanova, K. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv* **2018**, arXiv:1810.04805.
3. Bender, E.M.; Gebru, T.; McMillan-Major, A.; Shmitchell, S. On the Dangers of Stochastic Parrots: Can Language Models Be Too Big? In Proceedings of the 2021 ACM Conference on Fairness, Accountability, and Transparency, Virtual Event, 3–10 March 2021; pp. 610–623.
4. Polino, A.; Pascanu, R.; Alistarh, D. Model compression via distillation and quantization. *arXiv* **2018**, arXiv:1802.05668.
5. Zhu, M.; Gupta, S. To prune, or not to prune: Exploring the efficacy of pruning for model compression. *arXiv* **2017**, arXiv:1710.01878.
6. Liu, C.; Zoph, B.; Neumann, M.; Shlens, J.; Hua, W.; Li, L.J.; Fei-Fei, L.; Yuille, A.; Huang, J.; Murphy, K. Progressive neural architecture search. In Proceedings of the European Conference on Computer Vision (ECCV), Munich, Germany, 8–14 September 2018; pp. 19–34.
7. Howard, A.G.; Zhu, M.; Chen, B.; Kalenichenko, D.; Wang, W.; Weyand, T.; Andreetto, M.; Adam, H. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv* **2017**, arXiv:1704.04861.
8. Iandola, F.N.; Han, S.; Moskewicz, M.W.; Ashraf, K.; Dally, W.J.; Keutzer, K. SqueezeNet: AlexNet-level accuracy with 50× fewer parameters and <0.5 MB model size. *arXiv* **2016**, arXiv:1602.07360.
9. Google. Cloud Tensor Processing Units (TPUs). Available online: https://cloud.google.com/tpu/docs/tpus (accessed on 20 July 2025).
10. Intel. Intel® Vision Accelerator Design with Intel® Movidius™ Vision Processing Unit (VPU). Available online: https://www.intel.com/content/www/us/en/developer/topic-technology/edge-5g/hardware/vision-accelerator-movidius-vpu.html (accessed on 20 July 2025).
11. Pytorch. Pytorch. Available online: https://pytorch.org/ (accessed on 20 July 2025).
12. NVIDIA Corporation. NVIDIA TensorRT. Available online: https://developer.nvidia.com/tensorrt (accessed on 20 July 2025).
13. Microsoft. ONNX Runtime Documentation. 2023. Available online: https://onnxruntime.ai/docs (accessed on 20 July 2025).
14. Chen, T.; Moreau, T.; Jiang, Z.; Zheng, L.; Yan, E.; Shen, H.; Cowan, M.; Wang, L.; Hu, Y.; Ceze, L.; et al. TVM: An automated end-to-end optimizing compiler for deep learning. In Proceedings of the OSDI, Carlsbad, CA, USA, 8–10 October 2018.
15. GitHub, Inc. JAX: Composable Transformations of Python+NumPy Programs. 2023. Available online: https://github.com/google/jax (accessed on 20 July 2025).
16. NVIDIA. Jetson AGX Orin Technical Brief. 2023. Available online: https://developer.nvidia.com/embedded/jetson-agx-orin (accessed on 20 July 2025).
17. Cheng, Y.; Wang, D.; Zhou, P.; Zhang, T. A Survey of Model Compression and Acceleration for Deep Neural Networks. *arXiv* **2020**, arXiv:1710.09282.
18. Hao, T.; Huang, Y.; Wen, X.; Gao, W.; Zhang, F.; Zheng, C.; Wang, L.; Ye, H.; Hwang, K.; Ren, Z.; et al. Edge AIBench: Towards Comprehensive End-to-End Edge Computing Benchmarking. In Proceedings of the Benchmarking, Measuring, and Optimizing: First BenchCouncil International Symposium (Bench 2018), Seattle, WA, USA, 10–13 December 2018; Revised Selected Papers; Springer: Berlin/Heidelberg, Germany, 2018; pp. 23–30.

19. Shin, D.J.; Kim, J.J. Performance evaluation of YOLO models on embedded platforms using TensorRT. *Appl. Sci.* **2022**, *12*, 1154.
20. ONNX Runtime Developers. ONNX Runtime. Available online: https://onnxruntime.ai/ (accessed on 20 July 2025).
21. JAX Developers. JAX: High-Performance Array Computing. Available online: https://jax.readthedocs.io (accessed on 20 July 2025).
22. Ulker, B.; Stuijk, S.; Corporaal, H.; Wijnhoven, R. Reviewing inference performance of state-of-the-art deep learning frameworks. In Proceedings of the 23th International Workshop on Software and Compilers for Embedded Systems, St. Goar, Germany, 25–26 May 2020; pp. 48–53.
23. Wortsman, M.; Ilharco, G.; Gadre, S.Y.; Roelofs, R.; Gontijo-Lopes, R.; Morcos, A.S.; Namkoong, H.; Farhadi, A.; Carmon, Y.; Kornblith, S.; et al. Model soups: Averaging weights of multiple fine-tuned models improves accuracy without increasing inference cost. In Proceedings of the International Conference on Machine Learning (ICML), Baltimore, MD, USA, 17–23 July 2022.
24. Alqahtani, D.K.; Cheema, A.; Toosi, A.N. Benchmarking Deep Learning Models for Object Detection on Edge Computing Devices. *arXiv* **2024**, arXiv:2409.16808.
25. Yeom, S.K.; Kim, T.H. UniForm: A Reuse Attention Mechanism Optimized for Efficient Vision Transformers on Edge Devices. *arXiv* **2024**, arXiv:2412.02344.
26. Arya, M.; Simmhan, Y. Understanding the Performance and Power of LLM Inferencing on Edge Accelerators. *arXiv* **2025**, arXiv:2506.09554.
27. Stanford Vision Lab; Stanford University; Princeton University. IMAGENET. Available online: https://www.image-net.org/ (accessed on 20 July 2025).
28. He, K.; Zhang, X.; Ren, S.; Sun, J. Deep residual learning for image recognition. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, Las Vegas, NV, USA, 27–30 June 2016; pp. 770–778.
29. Sandler, M.; Howard, A.; Zhu, M.; Zhmoginov, A.; Chen, L.C. MobileNetV2: Inverted Residuals and Linear Bottlenecks. In Proceedings of the CVPR, Salt Lake City, UT, USA, 18–23 June 2018.
30. Tan, M.; Le, Q. EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks. In Proceedings of the ICML, Long Beach, CA, USA, 9–15 June 2019.
31. Liu, Z.; Lin, Y.; Cao, Y.; Hu, H.; Wei, Y.; Zhang, Z.; Lin, S.; Guo, B. Swin Transformer: Hierarchical Vision Transformer using Shifted Windows. In Proceedings of the ICCV, Montreal, QC, Canada, 11–17 October 2021.
32. Jocher, G. YOLOv5 by Ultralytics. 2020. Available online: https://github.com/ultralytics/yolov5 (accessed on 20 July 2025).
33. Frostig, R.; Johnson, M.; Leary, C. Compiling machine learning programs via high-level tracing. *Syst. Mach. Learn.* **2018**, *4*, 1–3.
34. Deng, J.; Dong, W.; Socher, R.; Li, L.J.; Li, K.; Li, F.-F. ImageNet: A Large-Scale Hierarchical Image Database. In Proceedings of the CVPR09, Miami, FL, USA, 20–25 June 2009.
35. Xu, R.; Han, F.; Ta, Q. Deep learning at scale on nvidia v100 accelerators. In Proceedings of the 2018 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS), Dallas, TX, USA, 12 November 2018; IEEE: Piscataway, NJ, USA, 2018; pp. 23–32.
36. Marcinkiewicz, P. ONNX Export Doubles RAM Usage. Available online: https://github.com/pytorch/pytorch/issues/61263 (accessed on 20 July 2025).
37. Karumbunathan, L. NVIDIA Jetson AGX Orin Series Technical Brief. Available online: https://www.nvidia.com/content/dam/en-zz/Solutions/gtcf21/jetson-orin/nvidia-jetson-agx-orin-technical-brief.pdf (accessed on 20 July 2025).
38. Foundation, T.L. Open Neural Network Exchange. Available online: https://onnx.ai/ (accessed on 20 July 2025).