Analog Integrated Circuits and Signal Processing (2026) 127:5
https://doi.org/10.1007/s10470-026-02580-8

Analyzing time and power efficiency of machine learning models on edge devices

Heba Khdr[1] · Mohamed Aboelenien Ahmed[1] · Yiğit Oğuz[1] · Jörg Henkel[1]

Received: 30 May 2025 / Revised: 2 March 2026 / Accepted: 13 March 2026

Abstract
Edge machine learning (EdgeML) refers to the execution of machine learning (ML) algorithms on devices located close to data sources. The primary goals of EdgeML are to reduce response time and preserve data privacy. Nevertheless, edge devices often face constraints in processing power, memory, and energy, making it challenging to deploy complex ML models, including neural networks (NNs). To address these limitations, significant efforts have focused on improving the time and power efficiency of EdgeML through model optimization and the use of hardware accelerators. Orthogonal to these efforts, this paper introduces EdgeMLProfiler, a novel open-source tool (https://gitlab.kit.edu/uexfc/EdgeMLProfiler), designed to evaluate the time and power consumption of training and inference processes for selected NNs across various hardware architectures and software libraries. Using EdgeMLProfiler, we present–for the first time–a comparative time and power analysis of several NN models, including widely used convolutional neural networks (CNNs) and custom-designed fully-connected neural networks (FNNs). Our analysis reveals distinct efficiency patterns across different models and hardware configurations, providing practical insights for selecting the most time- and power-efficient deployment configurations for ML models on edge devices.

Keywords Edge machine learning · Neural networks · Convolutional neural network · Inference time · Edge devices · Low power design

1 Introduction

The notion of edge machine learning (EdgeML) [1, 2] refers to the implementation of machine learning (ML) models directly on edge devices, such as smartphones, local servers, and embedded systems, rather than relying on cloud-based execution, as depicted in Fig. 1. EdgeML diminishes the necessity for transferring data between edge devices and cloud servers, thereby facilitating faster response times. This is vital for scenarios necessitating real-time data processing and instant decision-making. Another advantage of EdgeML is its ability to preserve data privacy by retaining data on edge devices. These advantages of EdgeML have made it appealing for a diverse array of applications, including autonomous vehicles, surveillance systems and healthcare [3–5].

Executing computation-intensive ML models, such as neural networks (NNs), on edge devices calls for efficient deployment to be compatible with edge devices, which typically suffer from limited computational resources, memory, and energy, compared to cloud servers [6, 7]. An *efficient ML deployment* on edge devices needs to meet the timing requirements of the given ML task while adhering to the device's power constraints. An ML deployment typically specifies both the hardware platform of the executing device and the software development framework. In terms of hardware, edge devices may incorporate various components such as CPUs and GPUs. Recently, GPUs have been integrated into embedded devices (e.g., [8]), as GPUs are generally more time-efficient than CPUs when executing NNs [9]. The performance of both CPUs and GPUs can vary significantly depending on their specific architectures. Regarding software, numerous ML development frameworks have emerged to support training and inference of NNs. A notable example is PyTorch [10], which has gained

✉ Heba Khdr
heba.khdr@kit.edu

[1] Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany

Published online: 26 March 2026

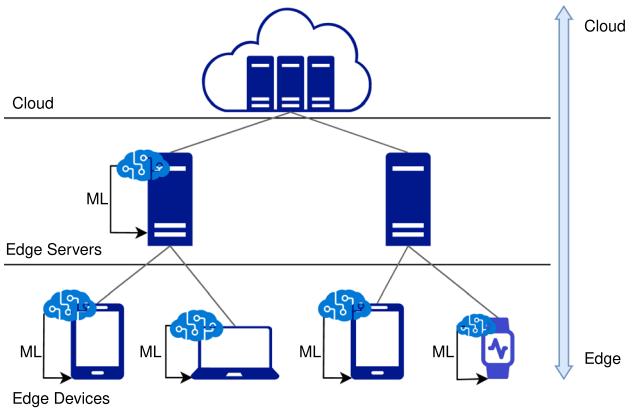

Fig. 1 In an EdgeML framework ML models are deployed on edge devices, which can include local servers, laptops, smartphones and smartwatches

widespread adoption in recent years [1] due to its features such as dynamic computational graphs, ease of debugging, and fast training performance. LibTorch, built on the same core as PyTorch, is a pure C++ library. Consequently, it is expected to be more time-efficient than PyTorch, which is constructed in the higher-level language Python.

To optimize for time and power efficiency of ML deployment, researchers have developed optimization techniques at both the hardware and software levels [11]. Methods like quantization and pruning [12] have been heavily used. Pruning removes certain parts of the model structure, such as nodes or branches in decision trees or neurons in NNs [13]. Quantization reduces the precision of numerical values. It is often applied to the weights and activation parameters of NNs [14]. These two approaches reduce model size, lower memory requirements, and improve computational efficiency, allowing ML models to operate effectively on edge devices. The two approaches are also considered for training on edge devices in Federated Learning (FL) settings [15, 16] to improve the computation and communication efficiency of FL systems. Another optimization technique is neural architecture search, which is used to design efficient neural networks to be deployed on edge devices given resource constraints such as memory and inference time [17]. Besides model optimization, researchers present efficient hardware accelerators, e.g., [18, 19]. Additionally, system-level techniques such as Dynamic Voltage and Frequency Scaling (DVFS) have been considered to enhance the efficiency of inference [20] and training [21] on edge devices.

To systematically evaluate machine learning performance on edge and embedded platforms, several standardized benchmarking suites have been proposed to assess efficiency across different tasks. MLPerf Tiny [22] is a standardized benchmark suite designed to evaluate machine learning performance on resource-constrained edge devices such as microcontrollers and embedded systems. It includes representative tasks reflecting real-world deployments (e.g., image classification and speech recognition), enabling the measurement of key metrics such as latency and energy consumption. This facilitates reproducible and fair comparisons of hardware–software co-design solutions for low-power edge AI applications. Similarly, AI Benchmark [23] evaluates on-device AI performance using a diverse set of practical workloads, including image processing, recognition, and neural network inference tasks, providing an application-level assessment of deep learning performance across mobile platforms. Profilers such as the Apache TVM and PyTorch profilers focus only on timing profiling deep learning model execution across heterogeneous hardware platforms.

Diverging from current studies focusing on enhancing EdgeML efficiency or benchmarking, we introduce EdgeMLProfiler, a novel tool for profiling *both the time and power efficiency* of various ML deployment setups, including software libraries (Pytorch, Libtorch) and hardware units (CPU, GPU). EdgeMLProfiler offers timing and power profiling for both training and inference of user-specified ML models. In this paper, we use our tool to present a comparative analysis of time and power efficiency across different NN deployments on diverse edge devices. Specifically, we implemented several convolutional neural networks (CNNs) and fully-connected neural networks (FNNs) due to their applications in a wide range of ML tasks. We performed experiments to evaluate both the training and inference times and the power consumption on the GPU and CPU of four actual devices using PyTorch and LibTorch. Our analysis reveals how different deployment decisions, such as hardware and software characteristics, affect time and power efficiency. Our tool enables developers to conduct this analysis on their NNs using their target hardware to choose the deployment option that optimally balances time and power efficiency according to their needs.

Our contributions:

- We introduce EdgeMLProfiler, an open-source tool that profiles the time and power required to execute different NNs on various real edge devices.
- We present a comparative analysis of the training/inference of various FNNs and CNNs that shows diverse time- and power-efficiency trends for different software libraries and hardware specifications.
- We demonstrate a case study exploring the effects of frequency downscaling on time and power consumption on two different devices for two CNN models and one FNN, highlighting the potential for more efficient deployments.

Table 1 The structure and input sizes for the employed CNNs

Network	Input Size	Convolutional layers	FC layers	Neurons per FC layer
LeNet	(1, 28, 28)	2	2	120, 84
AlexNet	(3, 224, 224)	5	2	4096, 4096
ResNet18	(3, 64, 64)	18*	1	512
VGGNet19	(3, 224, 224)	16	2	4096, 4096

*ResNet18 uses residual layers instead of traditional convolutional layers.

Table 2 The structure and input sizes for the employed custom FNNs

Network	Input size	FC layers	Neurons per layer
TinyNet	(1, 6)	1	30
SmallNet	(1, 24)	2	48
MediumNet	(1, 24)	3	100
BigNet	(1, 24)	6	1000

Table 3 The hardware specifications of the employed edge devices

	AMD-RTX	Jetson TX2 NX	Jetson Nano	Jetson Orin Nano
#CPU Cores	8	4	4	6
CPU Freq.	3700 MHz	2000 MHz	1479 MHz	1500 MHz
#CUDA Cores	10496	256	128	1024
GPU Freq.	1695 MHz	1300 MHz	921 MHz	918 MHz
GPU Memory	24 GB	4 GB*	4 GB*	8 GB*

*Shared with CPU.

- Our tool facilitates the selection of the most efficient NN deployment, which is compatible with the limited resources at the edge.

2 ML deployment setups

An ML deployment typically specifies both the hardware platform of the executing device and the software development framework. In this section, we illustrate our target ML models, hardware devices, and the software libraries.

2.1 ML models

The domain of ML encompasses an extensive array of models, each crafted for specific tasks and applications. Notably, FNNs and CNNs are widely used. Specifically, CNNs are frequently applied to image-related tasks, including image classification, object detection, and semantic segmentation [24, 25]. FNNs are primarily utilized for regression and classification tasks, as well as in deep reinforcement learning. This paper investigates time and power efficiency of deploying CNN and FNN models on different hardware architectures. To this end, we employ various CNNs and FNNs with varying architectures and input sizes. The selection of these network architectures covers a diverse set of tasks representative of real-world scenarios.

For CNNs, we employ four well-known networks which are: LeNet [26], AlexNet [24], ResNet18 [27], and VGGNet19 [28]. They differ from each other with the input size, numbers of convolutional layers and fully-connected (FC) layers, numbers of filters of each of layer. The details of the characteristics of these CNNs are shown in Table 1. While convolutional layers excel at capturing spatial patterns, fully connected layers are capable of modeling global relationships. The quantity of neurons in the fully connected layers grows in parallel with the convolutional layers' complexity. Additional attributes of the CNNs employed, including the number of filters per layer, kernel size, stride, and padding, adhere to the specifications set forth in the original models discussed in the previously mentioned papers.

For FNNs, we utilize four different architectures, ranging from a small FNN with a single hidden layer of 30 neurons to a large FNN with six hidden layers, each consisting of 1000 neurons. The architectural specifics of these FNNs are presented in Table 2. Increasing the number of hidden layers and neurons per layer can enhance the FNN's capacity to learn intricate patterns and features in the data, consequently improving accuracy and generalization performance. However, this also results in additional computations, thus extending the power and time required for inference. For both CNNs and FNNs, we use the Rectified Linear Unit (ReLU) activation function. ReLU is a popular activation function that introduces non-linearity, enabling the networks to model complex relationships and learn expressive representations from the data.

2.2 Hardware architectures

The hardware architectures of edge devices differ significantly in terms of computational capabilities and memory. Hence, the power and time efficiency of one ML model depends on the underlying hardware architecture. GPU is an important feature in modern devices and is known to be more time efficient than CPU in executing ML algorithms due to its parallel computational capabilities [9]. ML algorithms, especially deep learning algorithms, involve performing numerous repetitive computations, such as matrix multiplications and convolutions. GPUs excel at executing these computations in parallel as GPUs consist of thousands of cores that can work on different data points simultaneously.

In this paper, we consider four devices. Table 3 summarizes their specifications and shows their various capabilities. The first device is equipped with an AMD Ryzen 7 2700X CPU [29] and an Nvidia RTX 3090 GPU [30]. For simplicity, we will refer to this device as AMD-RTX. Among the

three, AMD-RTX boasts the highest computational power and can serve as a local server on the edge. The second device is the Nvidia Jetson TX2 NX [31]. The third device is Nvidia Jetson Nano [8]. The fourth device is Nvidia Jetson Orin Nano [32]. The Jetson devices qualify as embedded devices. These two devices also differ in their computational capabilities.

Each of the devices includes a GPU. The Nvidia RTX 3090 in the AMD-RTX is a top-tier graphics card that utilizes the Ampere architecture, containing 10,496 CUDA cores, which are specialized processors meant to enhance parallel computing performance. Its remarkable computational power makes it well-suited for AI-heavy applications such as deep learning training and inference, as well as for other demanding tasks like scientific simulations and data processing. With 24 GB of GPU memory, it effectively handles large datasets and intricate models. Conversely, the GPUs in the Jetson boards are components of Nvidia's edge computing platform, crafted for AI applications on small edge devices like embedded systems. The Jetson Nano is equipped with 128 CUDA cores, the Jetson TX2 NX offers 256 CUDA cores, whereas the Jetson Orin nano offers 1024 CUDA cores. These GPUs are engineered specifically for AI inference and edge computing applications, allowing for real-time data processing and analysis on the device itself. They offer ample capacity for running ML models and performing AI inference effectively. Their energy efficiency and compact form factor make them suitable for a variety of embedded AI applications, including robotics, autonomous systems, IoT devices, and smart cameras, empowering researchers to deploy AI capabilities at the network edge.

2.3 Software libraries

Several ML frameworks have emerged to facilitate ML training and inference. PyTorch [10] has become a prominent option for ML implementation, gaining considerable traction in academic research [33]. Experiments reported in [33] indicate that PyTorch offers faster training speeds than other widely-used frameworks, such as Tensorflow [34]. PyTorch uses dynamic computation graphs, which are used to construct the graph on-the-fly during the execution of the program, enabling it to handle variable-sized inputs and dynamic control flows, granting flexibility in development. PyTorch provides support for GPU accelerators, which allows for improved performance on supported hardware. Moreover, PyTorch is built to operate on a range of architectures, such as ARM-based systems, broadening its versatility and potential uses.

PyTorch's C++ counterpart, LibTorch [10], shares many of these benefits, since they are based on the same core [33]. Both PyTorch and LibTorch utilize shared libraries such as Autograd, ATen, and TorchScript. These shared libraries enable automatic differentiation, tensor operations, and just-in-time compilation, respectively, which enhance their capabilities and performance. Although programming with LibTorch may be more difficult than using the Python-based PyTorch, it benefits from being written in pure C++. This can theoretically enhance efficiency because C++ provides low-level capabilities and performance-focused features, enabling developers to maximize hardware utilization and optimize code. However, there is a lack of extensive comparison between these two libraries in terms of time efficiency for ML deployments. This paper makes this comparison for different CNN and FNN models while considering different hardware options for deployments. By conducting such a study, we can gain valuable insights into which library is better suited for EdgeML applications.

3 EdgeMLProfiler

EdgeMLProfiler is a lightweight tool for evaluating training and inference performance across different setups. It offers a user-friendly interface for comparing LibTorch and PyTorch, as well as CPU and GPU efficiency. Through its structured configuration, it aids in selecting the optimal framework for specific tasks. Use cases include assessing an embedded device's performance before deployment or identifying constraints to address before training or fine-tuning a model.

Open-Source Contribution: The source code for EdgeMLProfiler is available for download at https://gitlab.kit.edu/uexfc/EdgeMLProfiler and is released under MIT License for unrestricted use.

Figure 2 provides an overview of EdgeMLProfiler. The tool uses a JSON configuration file that includes all necessary parameters to build the desired NN model and outlines the deployment setup intended for testing. Based on the given configuration, the constructor module creates the corresponding network. Dependent on the chosen mode, the tool executes either the training module or the inference module and reports the average duration for one training step and one inference step. If power monitoring is enabled, the power measurement module is executed. The profiling results are then presented as output. Further details of the tool modules are provided below.

3.1 Verification of configuration file

EdgeMLProfiler accepts a configuration file in JSON format as input, which contains various parameters categorized as follows:

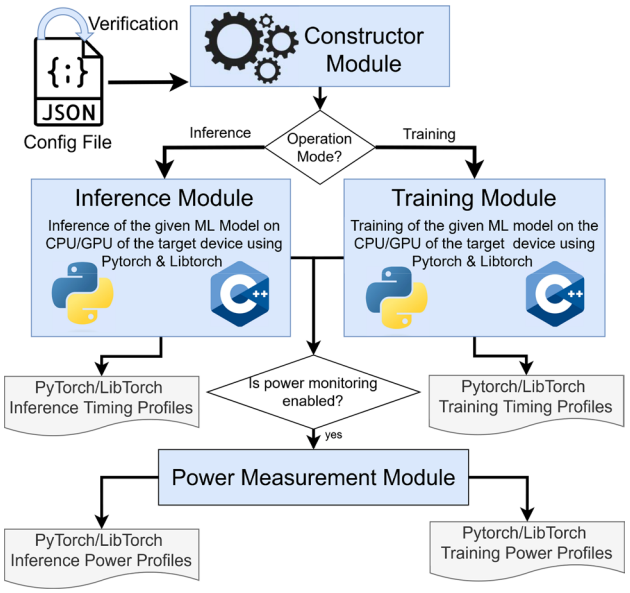

Fig. 2 An overview of our proposed tool, EdgeMLProfiler

- **General Parameters** cover the hardware execution unit (CPU or GPU), deployment mode (training or inference), and the ML model's task type (classification or regression).
- **Network Architecture** specify the NN input shape (width, height, number of channels), types and number of layers, along with the number of neurons in each layer.
- **Warm-up Parameters** refer to the total number of warm-up operations to be conducted.
- **Inference Parameters** specify the total number of forward passes to be executed.
- **Training parameters** include the optimizer type, learning rate, loss function, batch size, number of epochs, and training sample count.

Initially, EdgeMLProfiler reviews the configuration file to confirm its integrity. Specifically, it examines the syntax and structure of the file and ensures that all required settings or entries, along with their corresponding values, are present. Should any unauthorized value be detected in the configuration file, the tool halts its execution and provides feedback detailing the error.

3.2 Constructor module

EdgeMLProfiler accommodates a variety of neural network layer types, allowing users to build intricate network architectures suited to their particular edge computing needs. Every layer type supported has distinct features and parameters, offering users the versatility to create efficient, customized models for deployment on edge devices. Different layer types can be connected as long as their parameters are properly initialized.

The following layer types are supported by EdgeMLProfiler:

- Convolutional Layer (`'conv2d'`)
- Dense Layer (`'dense'`)
- Max Pooling Layer (`'maxpool2d'`)
- Flatten Layer (`'flatten'`)
- Batch Normalization Layer (`'batchnorm2d'`)
- Average Pooling Layer (`'averagepool2d'`)
- Dropout Layer (`'dropout'`)
- Residual Block (`'residual_block'`)

The chained layers are subsequently fed into the tool's respective constructor modules and connected in a cascading manner using a *sequential container* [35], which PyTorch/LibTorch allows for the easy construction of feed-forward models by stacking layers in a defined order.

3.3 Training/inference modules

EdgeMLProfiler performs preliminary operations to bring the system to a stable condition, ensuring accurate performance metrics before profiling. Specifically, the software runs a specified number of inference or training tasks (based on the mode) to warm-up the CPU, GPU, and caches. Users have the option to define the number of warm-up operations in the configuration file, thereby enabling the system to fine-tune and enhance performance prior to the start of formal profiling.

For inference mode, the tool executes the specified amount of forwards passes and outputs the average time taken for a forward pass as well as the total time.

For the training process, EdgeMLProfiler generates mock training data to simulate the training workload. The tool dynamically generates training samples based on the specified input/output shape, the task type, and number of training samples, providing a realistic simulation of the training process. Then, the tool simulates training with the generated mock training data. One training step consists of one forward pass, calculation of loss, then backpropagation to compute the gradients and update the model parameters. This process is repeated for a predefined number of epochs, with each epoch involving a full pass over the training data. The tool records the time taken for each epoch as well as the total time for all epochs. To determine the average time for a single training step (which includes the forward pass, loss calculation, and backward pass), we calculate it by dividing the epoch time by the result of the number of samples divided by the batch size, as specified in the training parameters.

3.4 Power measurement module

When enabled, this module tracks the real-time power usage of the device. It can be adjusted based on the device being utilized. For our embedded devices, specifically Nvidia Jetson boards, the tool employs the `jetson-stats` [36] utility, which is an open-source library offering a straightforward, unified interface for the monitoring and management of NVIDIA Jetson devices. Power values are logged at user-defined intervals and stored in a log file that includes metadata such as the configuration file name, operational mode (training or inference), device type (CPU or GPU), and a timestamp for traceability. Once profiling is complete, the module halts logging and trims the log file to align precisely with the start and end times reported by the inference or training module. It then computes the average power consumption over the active execution period, providing a more accurate measurement of the power usage during model operation.

4 Comparative efficiency analysis

To compare time and power efficiency of the target ML deployment setups presented in Section 2, we run our tool EdgeMLProfiler on the devices shown in Table 3. This profiling involves evaluating the performance of the training and inference of the ML models listed in Table 1 and Table 2. We use the four deployment options that specify the software framework, i.e., PyTorch and LibTorch, and the hardware execution units, i.e., CPU and GPU. Furthermore, we include time profiling for all devices and enable power measurements on the Jetson devices. This information is crucial for embedded devices because they are often used in battery-operated and energy harvesting settings, where power efficiency is of utmost importance. Furthermore, as an optional feature, the tool can be run in parallel with an additional workload that stresses the device, allowing these system-level effects to be naturally reflected in the profiling of training and inference measurements. However, all experiments in this evaluation are performed in isolation, without additional background workloads.

For each experiment, the given model is initialized with random values. Then, a predefined number of inference or training iterations (specified in the config file) are processed. The required times for conducting these iterations are recorded for each execution. Then, the mean of the recorded execution times is computed for each deployment configuration; for inference experiments, this corresponds to averaging over 5,000 inference runs. For power measurements, we additionally record the minimum, average, and maximum power values observed during execution. The experimental results of the inference processes are grouped according to the used device in Section 4.1, Section 4.2, Section 4.3, Section 4.4, and summarized in Section 4.5. Note that the inference times of different models might have different order of magnitude. Therefore, to enable comparing these inference time on a figure, we use two scales for the y-axis as shown in all of the comparison figures. The results of the training processes are discussed in Section 4.6.

4.1 Inference analysis on AMD-RTX

Figure 3 depicts the mean inference times of the selected models on AMD-RTX. For each model, four results are plotted to indicate the mean inference times of using PyTorch, LibTorch while running on CPU and GPU. Comparing first the inference times on CPU and GPU, we can observe that inference on GPU is faster in most of the cases, as expected. Comparing the efficiency of PyTorch and LibTorch implementations, we can observe that both PyTorch and LibTorch lead to similar inference times for the majority of the experiments, except for two scenarios where LeNet and ResNet18 are running on the GPU. These two scenarios show that LibTorch is slightly more efficient than PyTorch. Fig. 4 shows the results of the inference operations of our FNNs on

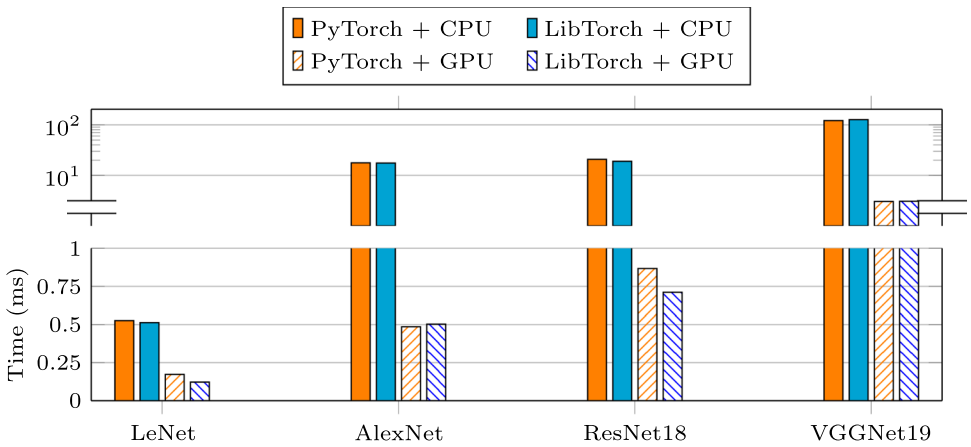

Fig. 3 Comparison of the inference times of CNN models on AMD-RTX, showing: 1) executing on GPU is always more efficient that executing on CPU, 2) there is no significant difference between using LibTorch or PyTorch, except for two scenarios, where LibTorch is slightly more efficient than PyTorch when LeNet and ResNet18 are executing on the GPU

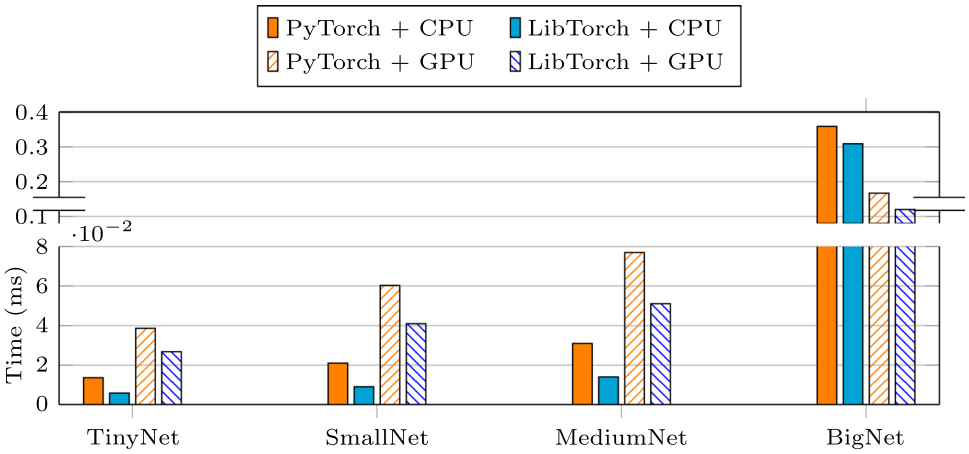

Fig. 4 Comparison of the inference times of FNN models on AMD-RTX, showing that 1) CPU is more efficient that GPU except for the BigNet 2) LibTorch is always more efficient than PyTorch

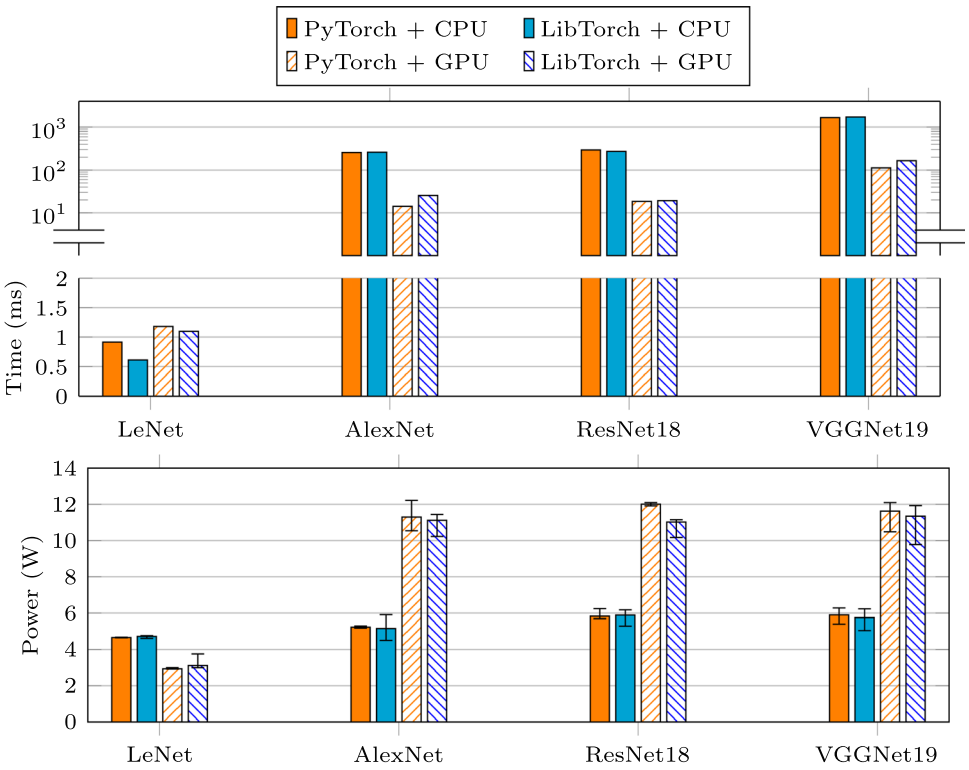

Fig. 5 Comparison of the inference times and power consumption of CNN models on Jetson TX2 NX. The figure displays the average values, while the whiskers for the power values indicate the maximum and minimum values. The inference time results show that 1) GPU is more efficient that CPU in most of the experiments, 2) LibTorch is more efficient than PyTorch only for the smallest model, LeNet

AMD-RTX. Unlike the results of CNNs, it is more efficient to perform the inference of FNNs on the CPU except for BigNet, as shown in Fig. 4. Moreover, LibTorch is always more efficient than PyTorch for FNNs. Hence, *GPU is more time efficient than CPU in executing CNNs models on AMD-RTX, while CPU is more efficient in executing relatively small FNNs. Comparing software libraries, LibTorch and PyTorch show similar efficiency in most cases of CNNs, while LibTorch is always more efficient than PyTorch for the used FNNs.*

4.2 Inference analysis on Jetson TX2 NX

Figure 5 has two sub-figures to depict the mean inference times and the power consumptions of the selected CNN models on Jetson TX2 NX. At the top sub-figure, four results are plotted for each model to indicate the mean inference times of using PyTorch, LibTorch while running on CPU and GPU. At the lower sub-figure, four results are plotted for each model to indicate the mean, max, min power consumption required for inference using PyTorch, LibTorch while running on CPU and GPU. Comparing first the time efficiency of inference on the CPU and GPU, we can observe that executing the models on GPU is faster in most of the cases, similar to the observed results on the powerful

device, i.e., AMD-RTX. Only for LeNet, CPU leads to a shorter inference time than GPU. Moreover, the CPU utilizes more power only for the LeNet inference, while the GPU as expected utilizes more power on the rest of CNN models.

Comparing PyTorch and LibTorch implementations, we can observe that only for LeNet, LibTorch leads to shorter inference times, while the remaining experiments show that PyTorch implementation is slightly faster and consumes a similar or slightly higher amount of power.

Fig. 6 shows the results of the inference operations of our FNNs on Jetson TX2-NX. Unlike the latency results of CNNs, it requires less latency and power to perform the inference of FNNs on the CPU except for BigNet. Moreover, LibTorch is always more efficient than PyTorch especially for the CPU execution. LibTorch also uses slightly more power than PyTorch, especially for the case of BigNet. In summary, we can observe that LibTorch is faster than PyTorch for smaller models and smaller input sizes.

4.3 Inference analysis on Jetson Nano

Figures 7 and 8 depict the mean inference times and power of the selected CNNs and FNNs on Jetson Nano, respectively. Comparing the efficiency of CNN models on CPU and GPU, we observe that GPU leads to faster inference times and higher average power in most experiments, except for LeNet, similar to the results on Jetson TX2 NX. Additionally, the efficiency in terms of power and time when employing LibTorch and PyTorch is comparable in the majority of scenarios, except for executing VGGNet19 on a GPU. Here, PyTorch operates more swiftly than LibTorch but results in increased power consumption. For FNN inference, CPU is more efficient in time and power than the GPU, except for the case of BigNet where the GPU is faster while using higher power. Additionally, for most FNNs, LibTorch is faster than PyTorch with comparable power use, except BigNet, where LibTorch consumes slightly more power than PyTorch. Fig. 9 shows power traces over time across different neural network when conducting multiple inferences with Pytorch. It indicates that, for CNNs, the power varies over time, whereas for FNNs it remains nearly constant.

4.4 Inference analysis on Jetson Nano Orin

Figures 10 and 11 depict the mean inference times and power of the selected CNNs and FNNs on Jetson Orin Nano, respectively. Comparing the efficiency of CNN models on CPU and GPU, we observe that GPU leads to faster inference times and higher average power in most experiments, except for LeNet. For this CNN, running on the GPU has less power consumption compared to running on the CPU. Additionally, the efficiency in terms of power and time

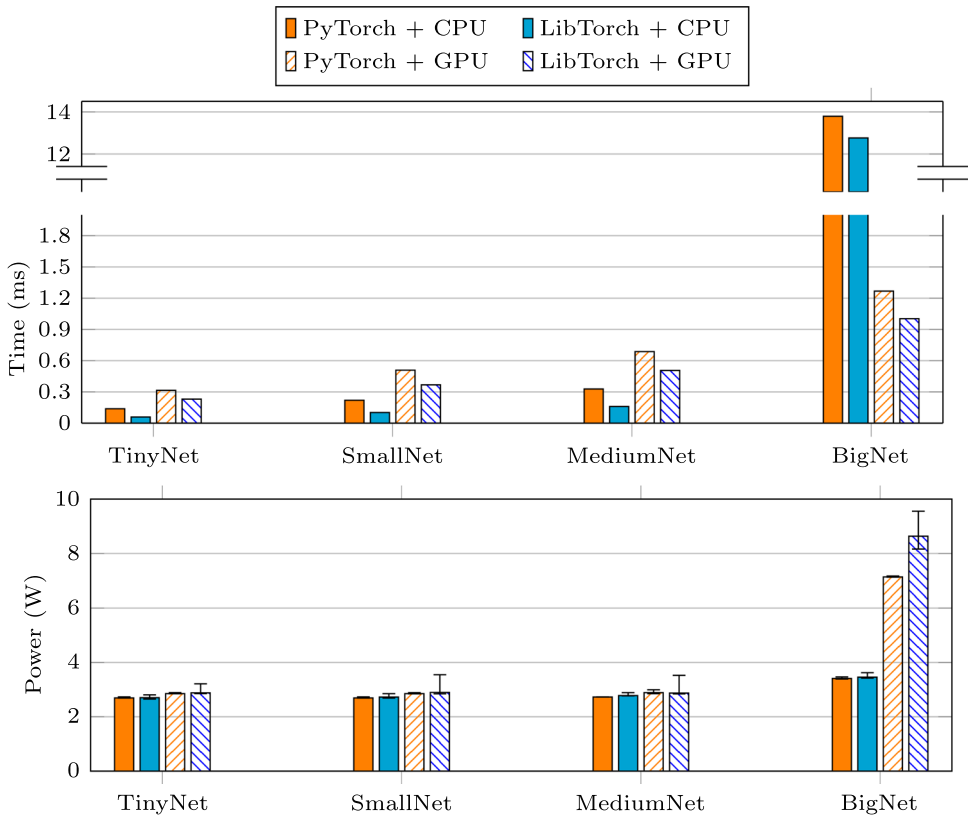

Fig. 6 Comparison of the inference times and and power consumption of FNN models on Jetson TX2 NX. The figure displays the average values, while the whiskers for the power values indicate the maximum and minimum values. The inference time results show that 1) CPU is efficient that GPU except for BigNet 2) LibTorch is always better than PyTorch especially with the CPU execution

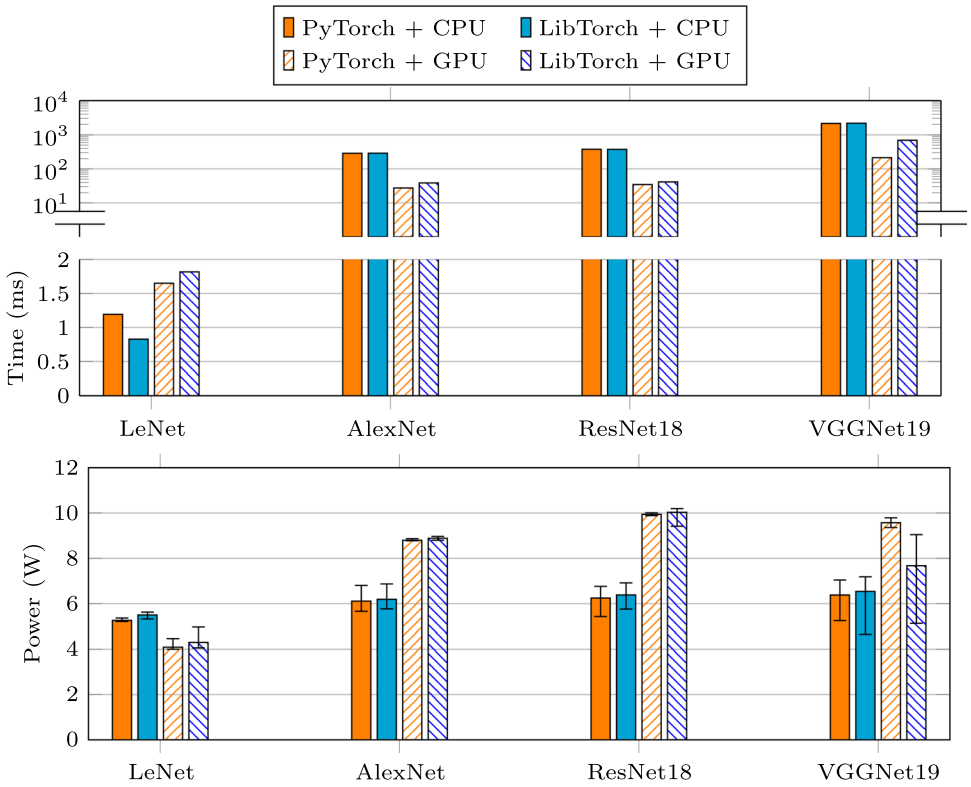

Fig. 7 Comparison of the inference times and power consumption of CNN models on Jetson Nano. The figure displays the average values, while the whiskers for the power values indicate the maximum and minimum values. The inference time results show that 1) GPU is more efficient, except for LeNet 2) LibTorch shows similar performance to PyTorch except for LeNet, that has faster inference using LibTorch on the CPU

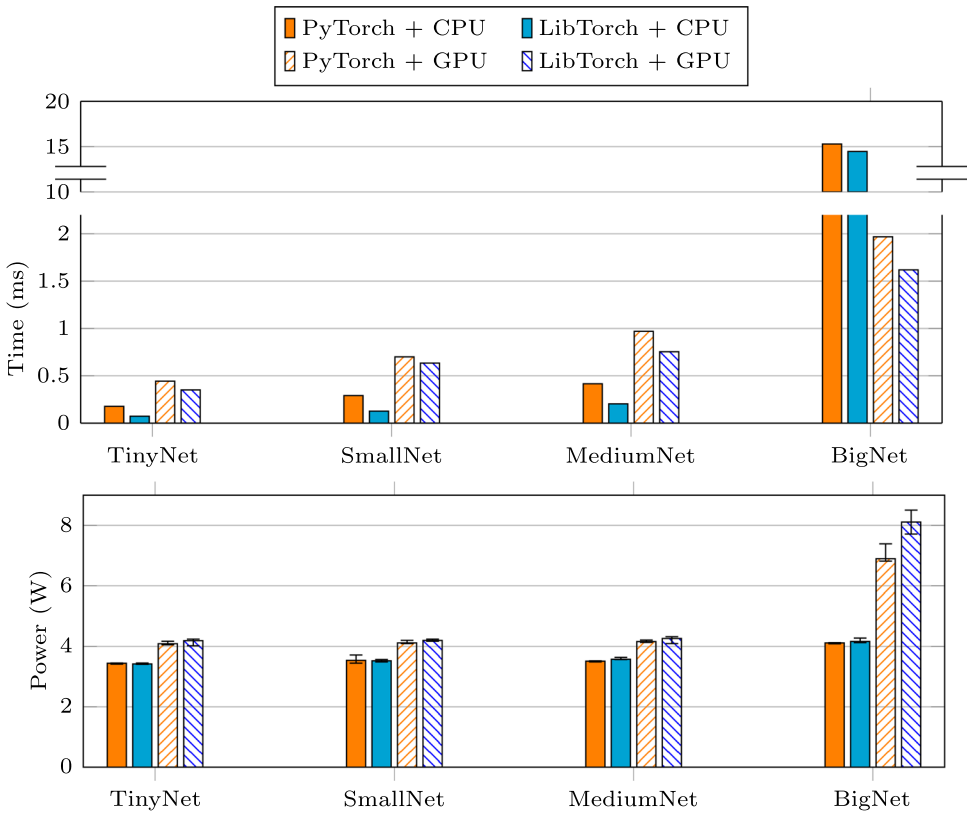

Fig. 8 Comparison of the inference times and power consumption of FNN models on Jetson Nano. The figure displays the average values, while the whiskers for the power values indicate the maximum and minimum values. The inference time results show that 1) CPU is more efficient than GPU except for BigNet 2) LibTorch is better than PyTorch in most of the cases

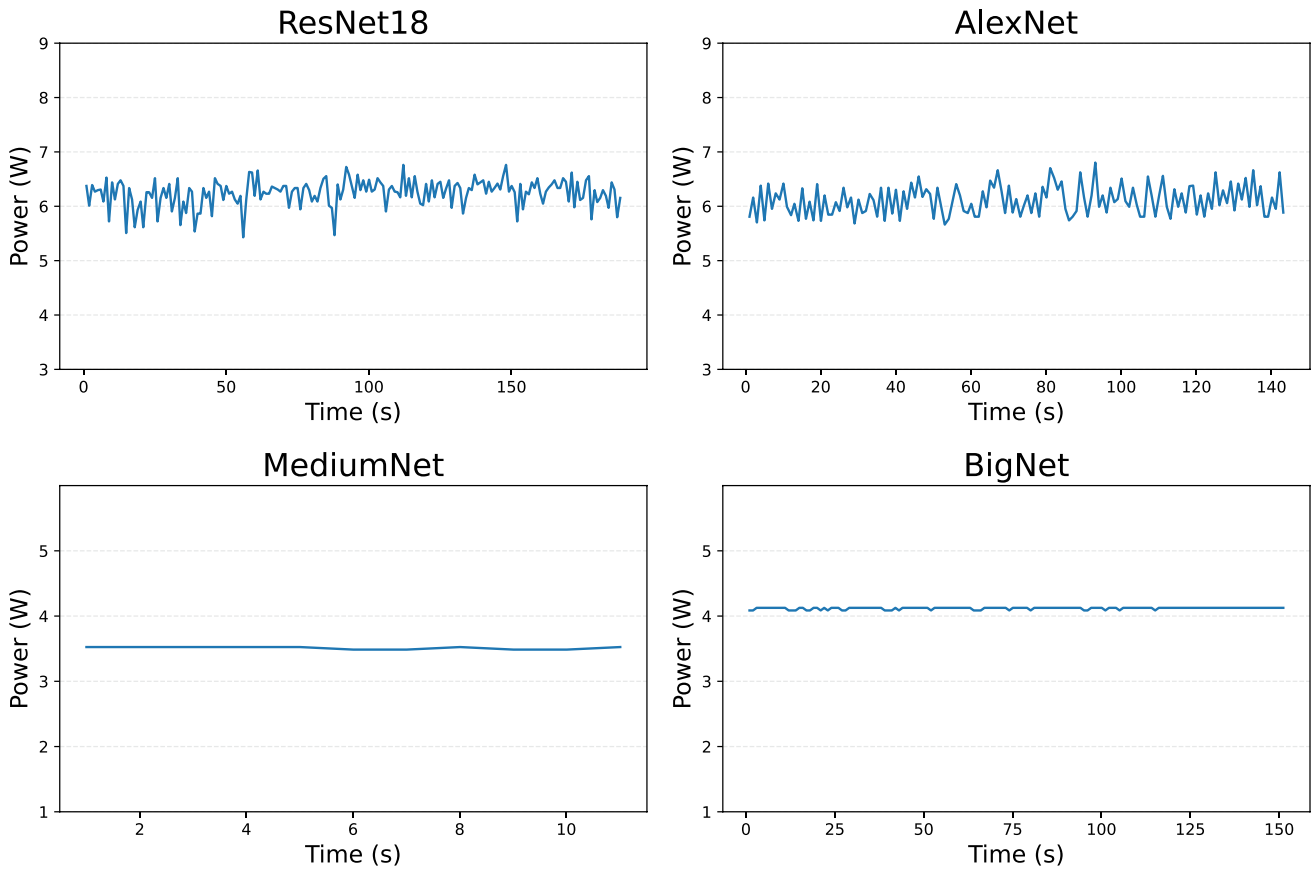

Fig. 9 Power traces over time across different neural network when conducting multiple inferences on Jetson Nano CPU with Pytorch. It indicates that, for CNNs, the power varies over time, whereas for FNNs it remains nearly constant

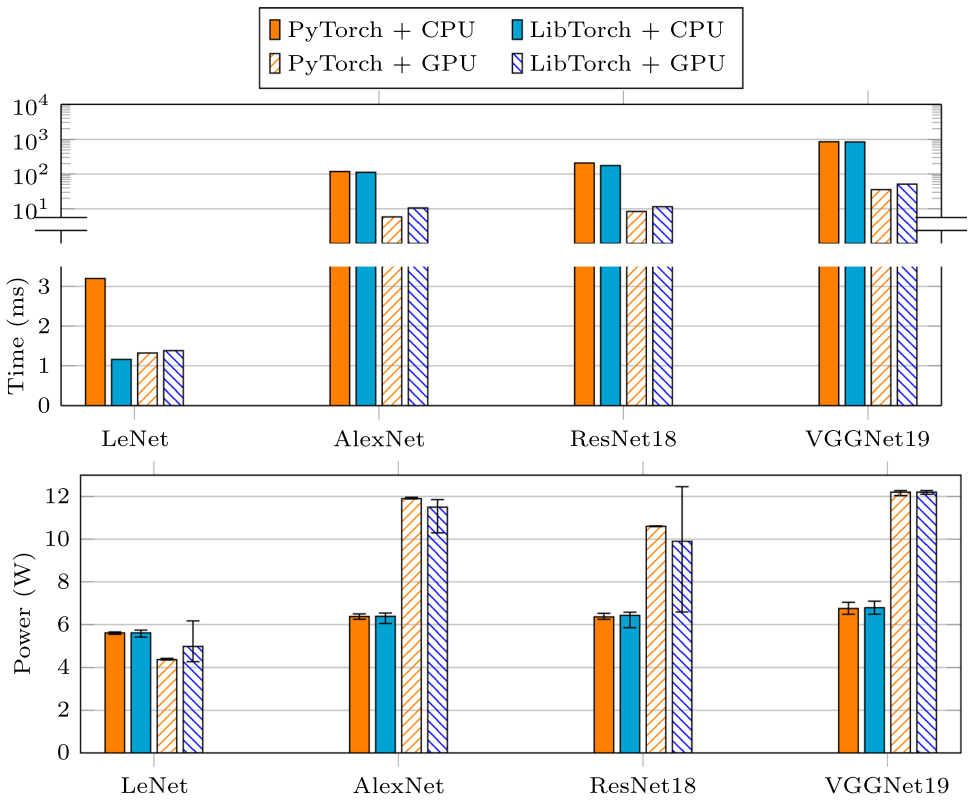

Fig. 10 Comparison of the inference times and power consumption of CNN models on Jetson Nano Orin. The figure displays the average values, while the whiskers for the power values indicate the maximum and minimum values. The inference time results show that 1) GPU is more efficient, except for LeNet 2) LibTorch shows similar performance to PyTorch except for LeNet, that has faster inference using LibTorch on the CPU

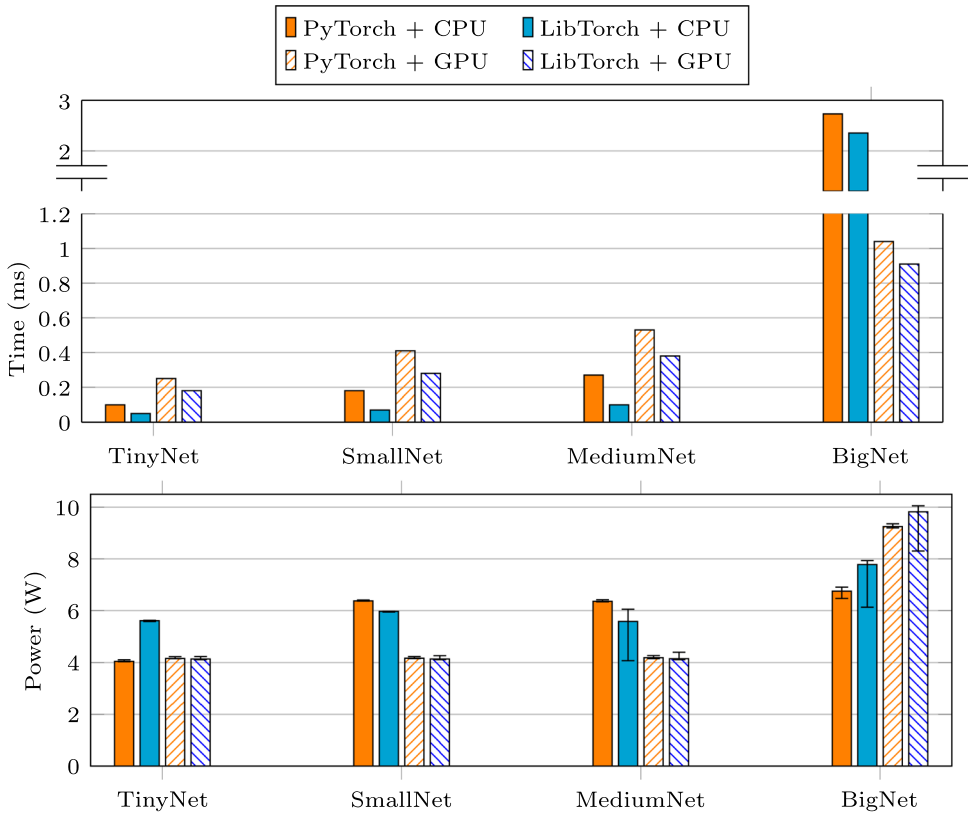

Fig. 11 Comparison of the inference times and power consumption of FNN models on Jetson Nano Orin. The figure displays the average values, while the whiskers for the power values indicate the maximum and minimum values. The inference time results show that 1) CPU is more efficient than GPU except for BigNet 2) LibTorch is better than PyTorch in most of the cases

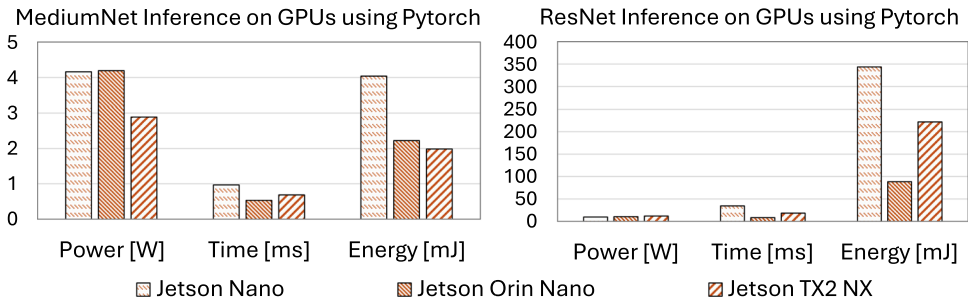

Fig. 12 Inter-device comparisons in terms of power, time and energy for MediumNet and ResNet

when employing LibTorch and PyTorch is comparable in the majority of scenarios. For FNN inference, CPU is more efficient in time and power than the GPU, except for the case of BigNet where the GPU is faster while using higher power. Moreover, for the majority of FNNs, LibTorch generally outperforms PyTorch in terms of speed while maintaining similar power consumption. However, LibTorch uses somewhat more power compared to PyTorch for BigNet and also for TinyNet on CPU.

4.5 Summary of inference results

Various devices exhibit distinct trends in the time efficiency of ML deployments. The most consistent trend observed across all tested devices is as follows: For large networks (CNNs and BigNet), GPUs consistently outperform CPUs in terms of time while recording higher average power, regardless of whether LibTorch or PyTorch is used. Importantly, the GPU's performance advantage over the CPU diminishes for smaller devices. In the case of smaller networks such as TinyNet, SmallNet, and MediumNet, CPUs exhibit better time and power efficiency compared to GPUs. The second noteworthy trend among the devices is that LibTorch exhibits better time efficiency at similar power consumption levels compared to PyTorch for all employed FNNs, which are deemed smaller models relative to the employed CNNs.

Figure 12 shows power, time and energy comparison between the three embedded devices, i.e., Jetosn TX2 NX, Jetson Nano, and Jetson Orin Nano. This comparison considers one CNN, ResNet, and one FNN, MediumNet, while they are running on the GPU of these devices. It is important to note that a similar comparison can be carried out on other CNNs and FNNs by examining the specific results across

the various figures that pertain to the different devices. It can be observed from this figure that Jetson Nano has led to the highest energy.The rationale is that it is the slowest of the three devices when power consumption is comparable. This is primarily due to its GPU being substantially smaller compared to those in other devices, resulting in longer execution times. However, the total power required to perform the same computations was comparable. This does not imply that this device always exhibits the lowest energy efficiency among all devices. A counterexample (see Fig. 8 and Fig. 11) is the inference of TinyNet on the CPU of the Jetson Nano, which is more energy efficient than that on the Jetson Orin Nano. In a comparison between Jetson Orin and Jetson TX2, it is noted that Jetson Orin exhibits greater energy efficiency (60% less energy) when running ResNet, whereas Jetson TX2 demonstrates marginally better efficiency (10% less energy) when executing MediumNet. These inter-device comparisons indicate that there is no definitive answer as to which device is the most efficient. Therefore, conducting comprehensive analyses prior to deployment is necessary to choose the optimal option, a process facilitated by our tool, EdgeMLProfiler.

Table 4 Training duration per training step and power consumption on various CNNs on GPU. Note that VGGNet19 results are unavailable for some devices due to memory constraints

Training Duration					
Device	Framework	LeNet	AlexNet	ResNet18	VGGNet19
AMD-RTX	PyTorch	2 ms	38 ms	76 ms	526 ms
	LibTorch	1.5 ms	34 ms	71 ms	486 ms
Jetson TX2 NX	PyTorch	13.6 ms	2079.9 ms	344.7 ms	-
	LibTorch	10.6 ms	3240.4 ms	360.9 ms	-
Jetson Nano	PyTorch	18.4 ms	2817.7 ms	1260.8 ms	-
	LibTorch	14.4 ms	3812.2 ms	2641.8 ms	-
Jetson Orin Nano	PyTorch	12.2 ms	330 ms	577 ms	-
	LibTorch	8.9 ms	471.5 ms	588 ms	-
Average Power Consumption					
Device	Framework	LeNet	AlexNet	ResNet18	VGGNet19
Jetson TX2 NX	PyTorch	4.2 W	4.9 W	9.4 W	-
	LibTorch	4.6 W	5.7 W	10.4 W	-
Jetson Nano	PyTorch	4.8 W	8.8 W	7.0 W	-
	LibTorch	5.8 W	7.7 W	7.1 W	-
Jetson Orin Nano	PyTorch	4.68 W	10.1 W	12.21 W	-
	LibTorch	5.91 W	9.42 W	12.33 W	-

4.6 Training analysis

The CNNs and FNNs models were trained using the devices' GPUs. Importantly, training the largest model, VGGNet19, on smaller edge devices like the Jetson TX2-NX, Jetson Nano, and Jetson Orin Nano was impractical due to their insufficient memory capacity, which does not satisfy the substantial memory demands required for VGGNet19 training. In Table 4, we record the average power and training time per step for CNNs. When comparing the time efficiency of LibTorch and PyTorch, it becomes evident that LibTorch offers notably higher efficiency for small models such as LeNet from CNNs. On the other hand, Pytorch shows lower latency on the three Jetson boards over AlexNet and ResNet18. Furthermore, Pytorch uses less average power on the Jetson devices.

Table 5 shows the average power and training time per step for all FNNs. LibTorch results in lower latency and higher power for all FNNs. The latency improvement with LibTorch is particularly notable on small edge devices such as Jetson TX2-NX and Jetson Nano.

5 Case study: voltage & frequency scaling

Our EdgeMLProfiler can be used to test the impact of system-level optimization techniques. In this section, we provide a case study to assess the impact of voltage & frequency scaling (VFS) to tradeoff power and time. Note that on our used boards, the users can only adjust frequency values, and the corresponding voltage will be automatically set. Therefore, we will report in this example, the frequency values.

Firstly, the user can adjust the frequency (and voltage) of the processor, then utilize our EdgeMLProfiler to measure the target NN inference time and power under such a setting. We selected two CNN examples; AlexNet and LeNet, and one FNN example, which is BigNet. In this case study we use two boards; Jetson Nano and Jetson Orin Nano. Table 6 shows the considered frequency levels for the CPU and the GPU of these boards used in this case study.

5.1 VFS analysis for AlexNet

For the first example, AlexNet, we observe from Fig. 7 that executing this network on Jetson Nano using the GPU requires significantly less inference time compared to the CPU, but at the cost of power. To reduce the power consumption, for the GPU deployment, we downscale the frequency levels of the GPU, and use our tool to measure the

Table 5 Training duration per training step and power consumption on various FNNs on GPU

Training Duration					
Device	Framework	TinyNet	SmallNet	MediumNet	BigNet
AMD-RTX	PyTorch	0.8 ms	0.9 ms	1.1 ms	1.9 ms
	LibTorch	0.8 ms	0.9 ms	1.3 ms	1.9 ms
Jetson TX2 NX	PyTorch	7.5 ms	8.6 ms	10.4 ms	25.1 ms
	LibTorch	3.6 ms	5.0 ms	6.3 ms	21.4 ms
Jetson Nano	PyTorch	9.9 ms	12.0 ms	14.1 ms	46.1 ms
	LibTorch	5.1 ms	6.7 ms	8.6 ms	44.4 ms
Jetson Orin Nano	PyTorch	5.4 ms	6.8 ms	8.15 ms	15.3 ms
	LibTorch	3.45 ms	4.45 ms	5.1 ms	11.5 ms
Average Power Consumption					
Device	Framework	TinyNet	SmallNet	MediumNet	BigNet
Jetson TX2 NX	PyTorch	2.7 W	2.8 W	2.9 W	7.9 W
	LibTorch	2.9 W	2.9 W	3.1 W	9.3 W
Jetson Nano	PyTorch	4.0 W	4.0 W	4.1 W	7.7 W
	LibTorch	4.3 W	4.3 W	4.3 W	7.7 W
Jetson Orin Nano	PyTorch	4.15 W	4.17 W	4.27 W	9.12 W
	LibTorch	5.23 W	5.4 W	5.65 W	10.98 W

Table 6 The frequency levels in MHz used in the case study

	f_1	f_2	f_3	f_4	f_{max}
Jetson Nano GPU	307	460	614	768	921
Jetson Nano CPU	920	1130	1320	-	1470
Jetson Orin Nano GPU	306	408	510	612	642
Jetson Orin Nano CPU	1110	1270	1420	-	1510

*Shared with CPU.

resulting power and time on Jetson Nano as provided in Fig. 13. Additionally, we show the CPU values at the maximum frequency f_{max}. We can notice that running the inference at f_3 (or lower frequency) results in similar (or smaller) power values than executing on the CPU, but the time required for the inference is still much less than the time on the CPU, as it can be observed from the time plot. Comparing only between executing on GPU at different frequencies, we can notice some scenarios where adjusting the configuration yields a better time-power tradeoff. For example, in the GPU deployment using LibTorch, reducing the GPU frequency from f_5 to f_4 can achieve a 12.4% reduction in average power with only 0.25% increase on inference time.

Figure 14 shows the same analysis on Jetson Orin Nano. As observed with the Jetson Nano, executing AlexNet on a GPU significantly decreases the execution time, while it results in increased power consumption. For this device, we observe that even when the frequency of the GPU is minimized, it continues to consume more power than when executing on the CPU. Nonetheless, the execution time on the GPU is an order of magnitude shorter compared to the CPU. This makes the GPU is the most efficient solution for this network on this device. Furthermore, it is evident that the power consumption of Jetson Orin Nano exceeds that of the Jetson Nano. Nevertheless, operating Jetson Orin at a lower frequency can lead to equivalent power usage as when Jetson Nano runs at its maximum frequency, and still with reduced time latency. Specifically, deploying PyTorch with a GPU on the Jetson Nano at f_{max} results in approximately 8 W power consumption and a latency of about 28 ms. Conversely, PyTorch with GPU deployment on the Jetson Orin at f_1 maintains around 8 W power usage, but the execution time is reduced to roughly 6 ms, which is 78% faster compared to the quickest option available on the Jetson Nano. Consequently, running AlexNet is consistently more efficient on the Jetson Orin board compared to the Jetson Nano.

5.2 VFS analysis for LeNet

As shown before in Fig. 7 that the CPU deployment for LeNet is more time efficient than the GPU, but it leads to higher power consumption. Therefore, we present the impact of downscaling frequency levels of the CPU in Fig. 15. Additionally, we show the GPU values at the maximum frequency f_{max}.

First of all, we can see that downscaling the frequency of the CPU to f_2 (or lower) results in similar or lower power consumption than the GPU deployment at f_{max}, while resulting lower latency when LibTorch is used. Furthermore, we have observed before in Fig. 7, Libtorch is more time-efficient than pytorch, but the former uses more average power. In Fig. 15, we show that by downscaling the CPU frequencies, it is possible for Libtorch on CPU to achieve comparable or less power than Pytorch at the maximum frequency while still sustaining lower latency. In particular, lowering the frequency from f_4 to f_3 would result in significantly reduced latency for Pytorch at the maximum frequency (f_{max}), while also slightly decreasing power consumption.

Figure 16 illustrates that power consumption for LeNet deployments using PyTorch and LibTorch on the CPU is nearly the same, whereas LibTorch deployment demonstrates

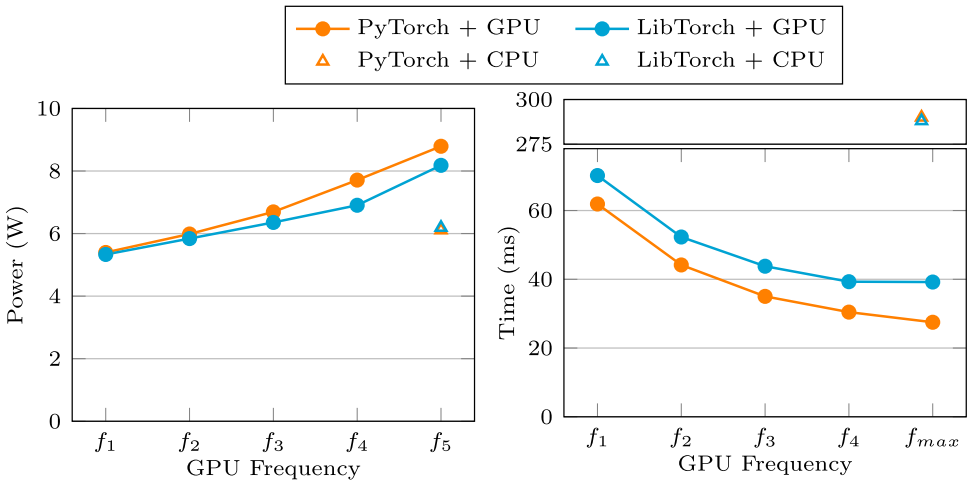

Fig. 13 Inference Time and Average Power Consumption at Different GPU Frequencies for AlexNet on Jetson Nano. Using our profiling tool, it is possible to evaluate GPU frequencies and select frequencies that balance time and power trade-offs. As demonstrated in the Libtorch example, reducing the frequency from f_5 to f_4 offers power savings with negligible differences in latency. Furthermore, the GPU can also use lower power by reducing frequency than the CPU at the maximum frequency and still better provide performance

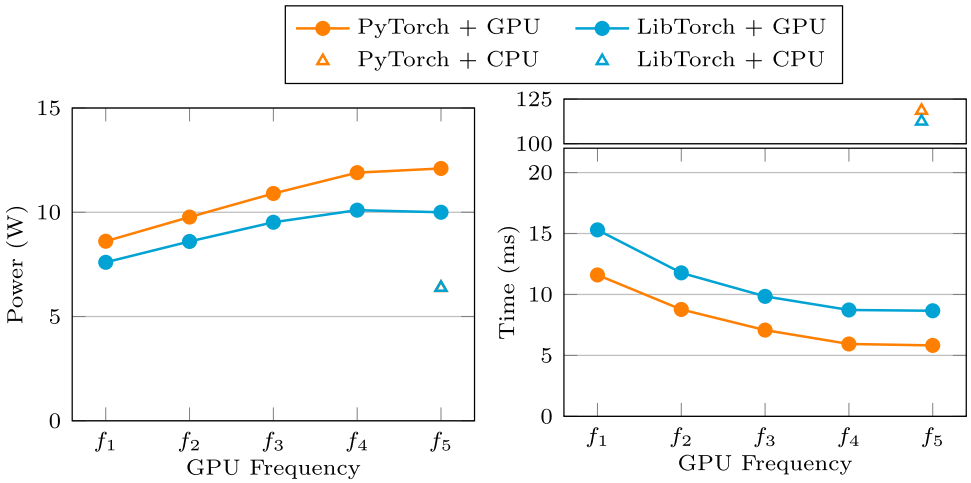

Fig. 14 Inference Time and Average Power Consumption at Different GPU Frequencies for AlexNet on Jetson Orin Nano. It is notable that when the GPU operates at its minimum frequency, the power consumption remains higher than when the CPU functions at its maximum frequency. Nonetheless, the execution time on the GPU is an order of magnitude shorter compared to the CPU. This makes the GPU is the most efficient solution for this network on this device

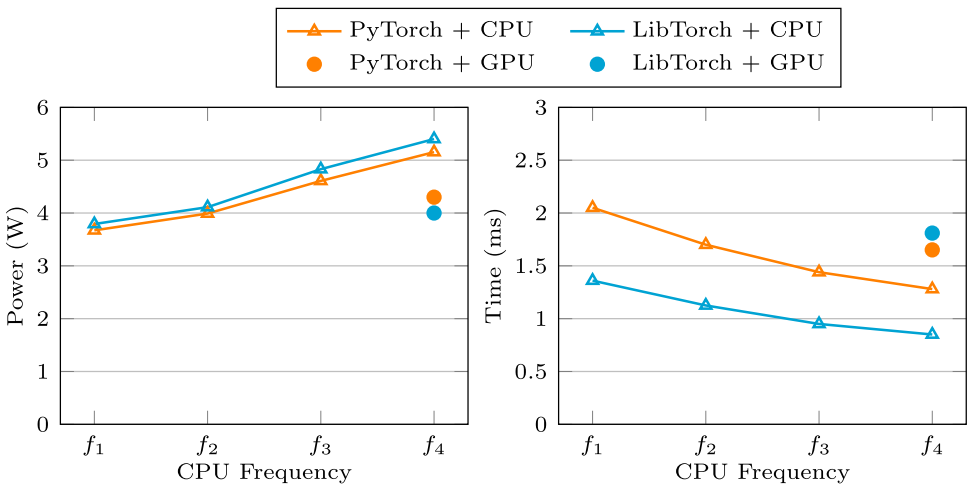

Fig. 15 Inference Time and Average Power Consumption at Different CPU Frequencies for LeNet5 on Jetson Nano. PyTorch tends to consume less power at the cost of higher latency. However, by lowering the CPU frequency when using Libtorch f_3 or f_2 it is possible to achieve both reduced power consumption and lower latency to Pytorch at the highest frequency (f_4). Furthermore, by reducing CPU frequency it is possible to get less power than the GPU with maximum frequency, while still providing lower latency

a significantly reduced execution time. LibTorch proves to be more efficient than PyTorch on the CPU for this network. Deploying PyTorch on the GPU at maximum frequency emerges as the most efficient solution, exhibiting reduced time and power consumption.

5.3 VFS analysis for BigNet

As illustrated in Figs. 8 Fig. 11, BigNet demonstrates more time-efficient deployment on a GPU compared to a CPU, although this comes at the expense of increased power consumption. In Fig. 17, we illustrate the effect of VFS on

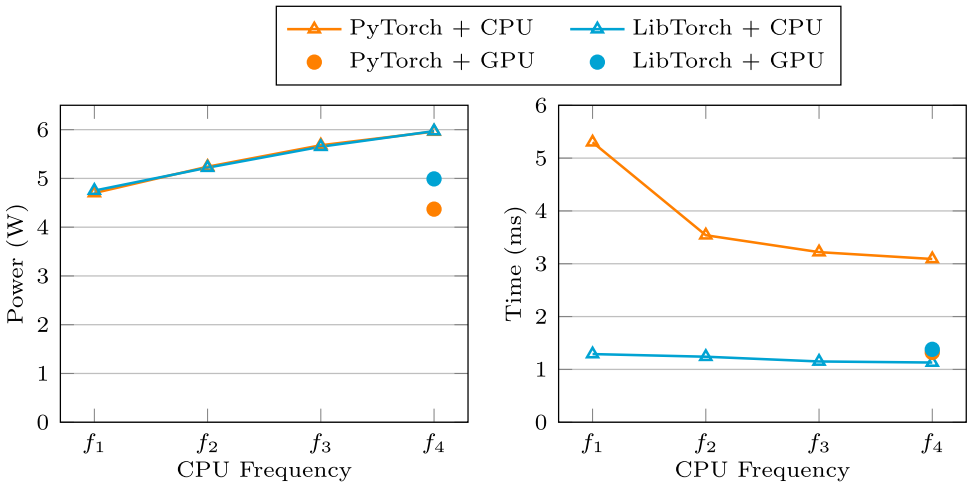

Fig. 16 Inference Time and Average Power Consumption at Different CPU Frequencies for LeNet5 on Jetson Orin Nano. PyTorch tends to consume less power at the cost of higher latency. However, by lowering the CPU frequency when using Libtorch f_3 or f_2 it is possible to achieve both reduced power consumption and lower latency to Pytorch at the highest frequency (f_4). Furthermore, by reducing CPU frequency it is possible to get less power than the GPU with maximum frequency, while still providing lower latency

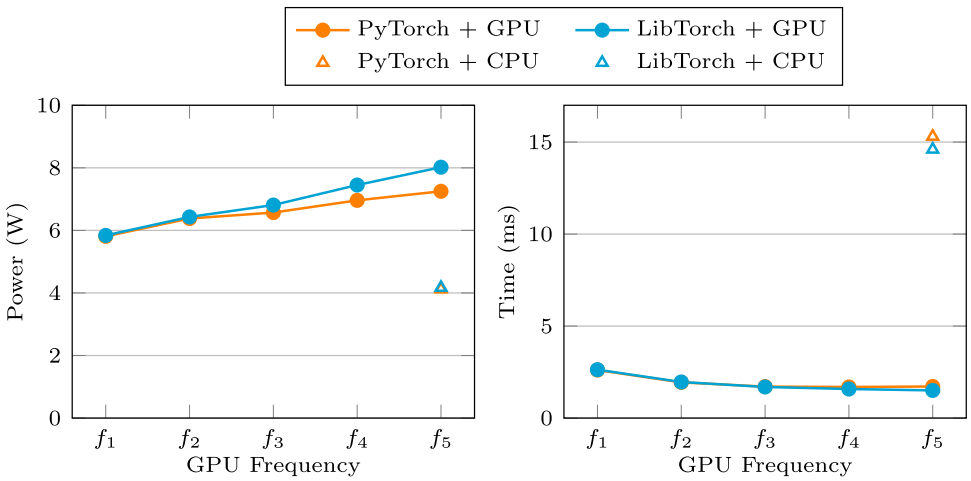

Fig. 17 Inference Time and Average Power Consumption at Different GPU Frequencies for BigNet on Jetson Nano. Using our profiling tool, it is possible to evaluate GPU frequencies and select frequencies that balance time and power trade-offs. As demonstrated in the Libtorch example, reducing the frequency from f_5 to f_4 offers power savings with negligible differences in latency. However, the GPU will always use higher power compared to the CPU, even if its frequency has been reduced to the minimum level

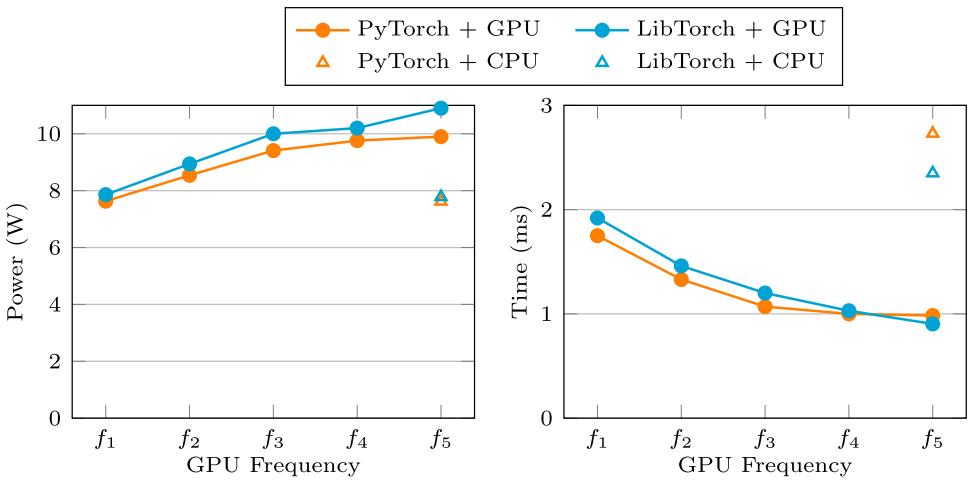

Fig. 18 Inference Time and Average Power Consumption at Different GPU Frequencies for BigNet on Jetson Orin Nano. Reducing the GPU frequency to its minimum level resulted in power consumption comparable to that of the CPU while achieving an 18% reduction in execution time

the Jetson Nano GPU concerning both time and power. It is evident that even when the frequency is reduced to the minimum level, the GPU still consumes more power than the CPU. Conversely, downscaling on the Jetson Orin offers improved trade-offs as shown in Fig. 18. Specifically, reducing to the minimum frequency results in similar power consumption to the CPU, while time efficiency remains 18% better. Notably, for the AlexNet analysis, we observe an opposite trend where downscaling the GPU on the Nano provides advantages, in contrast to the Orin, which does not.

In summary, this case study further complements the selection of the optimal ML deployment that has the best

tradeoff of time and power efficiency. The assessment of various deployment options is enabled by our tool.

6 Conclusion

We introduced EdgeMLProfiler, a new open-source tool for analyzing the time and power efficiency of ML models. Utilizing our tool, we conducted a comparative study of different CNNs and FNNs on actual edge devices with respect to time and power efficiency. Our analysis revealed trends indicating which combinations of hardware and software maximize efficiency for specific model structures and input sizes. These findings inform researchers and developers about the parameters impacting time and power efficiency, aiding them in selecting the best specifications for their ML applications. This tool's modular design facilitates the easy incorporation of various ML models, like recurrent neural networks (RNNs) and transformer models, into the Torch framework, enhancing both analyses and applications. Moreover, if the devices feature other processing components apart from the CPU and GPU, integrated with the processor, such as ASIC accelerators, e.g., deep learning accelerator (DLA), or FPGA-MPSoCs, e.g., ZynQ chip from Xilinx, the tool could also be employed to profile the training and inference duration of executing the ML models on these components. To conduct power profiling, the power module of our tool should be extended to incorporate functionalities for reading sensors on these devices, assuming they possess power sensors. In conclusion, this paper paves the way for further detailed investigations to uncover more possibilities for implementing ML on edge devices.

6.1 Potential tool extensions

Our open-source tool can be extended to support more deployment options. Currently, it focuses on CNNs and FNNs within the PyTorch/LibTorch frameworks. It would be beneficial for future extensions to support other frameworks such as TensorFlow Lite, ONNX Runtime, and TensorRT. Furthermore, integrating advanced model optimization methods–such as quantization, sparse pruning, and compression–as well as deployment-focused techniques like model partitioning or offloading would allow for a more thorough assessment of efficiency–accuracy trade-offs in edge AI systems. Finally, the framework can be expanded to consider various execution scenarios, such as multi-task execution, multi-threaded workloads, and real-time scheduling scenarios.

Acknowledgements A preliminary version of this work has been published in ICM 2024 [37]. Our tool, first introduced at ICM, was primarily developed for time profiling of ML models. In our journal article, we have enhanced its functionality by incorporating a new module specifically for power profiling. To this end, additional experiments were conducted to generate power profiles for CNN and FNN models across all devices, employing every combination of LibTorch, PyTorch, GPU, and CPU. These experimental results are presented in new visual plots. Furthermore, we performed experiments with a new edge device, the Jetson Orin Nano, with detailed discussions of the results provided in a new section along with associated figures. Finally, we examine the effects of voltage and frequency scaling (VFS) on two CNN and one FNN model across two boards, demonstrating VFS as an effective technique for optimizing the power-performance tradeoff in ML deployments.

Author Contributions H.K. is the main author; they conceived the original idea, supervised the student work, and wrote the majority of the paper. M.A. contributed to the extension of the tool and helped write the section on the power measurement module and helped in both revisions. Y.O. is the student who carried out the implementation. J.H. contributed to the idea discussions, improved the writing, and participated in the review process.

Funding Open Access funding enabled and organized by Projekt DEAL. This work was partially funded by the Federal Ministry of Research, Technology, and Space, BMFTR, as part of the DI-EDAI project (grant: 16ME0990K).

Data Availability No datasets were generated or analysed during the current study.

Declarations

Competing interests The authors declare no competing interests.

References

1. Murshed, M. S., Murphy, C., Hou, D., Khan, N., Ananthanarayanan, G., & Hussain, F. (2021). Machine learning at the network edge: A survey. *ACM Computing Surveys (CSUR), 54*(8), 1–37.
2. Rustemli, S., Alani, A. Y. B., Şahin, G., & Sark, W. (2025). Action detection of objects devices using deep learning in iot applications. *Analog Integrated Circuits and Signal Processing, 123*(1), Article 5.
3. Zebin, T., Scully, P. J., Peek, N., Casson, A. J., & Ozanyan, K. B. (2019). Design and implementation of a convolutional neural network on an edge computing smartphone for human activity recognition. *IEEE Access, 7*, 133509–133520.

4. Fernández-Sanjurjo, M., Mucientes, M., & Brea, V. M. (2021). Real-time multiple object visual tracking for embedded gpu systems. *IEEE Internet of Things Journal, 8*(11), 9177–9188.
5. Ganesh, P., Chen, Y., Yang, Y., Chen, D., & Winslett, M. (2022). Yolo-ret: Towards high accuracy real-time object detection on edge gpus. In: Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision, pp. 3267–3277.
6. Zhao, T., Xie, Y., Wang, Y., Cheng, J., Guo, X., Hu, B., & Chen, Y. (2022). A survey of deep learning on mobile devices: Applications, optimizations, challenges, and research opportunities. *Proceedings of the IEEE, 110*(3), 334–354.
7. Pfeiffer, K., Rapp, M., Khalili, R., & Henkel, J. (2023a). Federated learning for computationally constrained heterogeneous devices: A survey. *ACM Computing Surveys, 55*(14s), 1–27.
8. Nvidia: Jetson Nano. https://www.nvidia.com/de-de/autonomous-machines/embedded-systems/jetson-nano/
9. Schlegel, D. (2015). Deep machine learning on gpu. University of Heidelber-Ziti **12**.
10. Meta AI: PyTorch. https://pytorch.org/
11. Shuvo, M. M. H., Islam, S. K., Cheng, J., & Morshed, B. I. (2022). Efficient acceleration of deep learning inference on resource-constrained edge devices: A review. *Proceedings of the IEEE, 111*(1), 42–91.
12. Liang, T., Glossner, J., Wang, L., Shi, S., & Zhang, X. (2021). Pruning and quantization for deep neural network acceleration: A survey. *Neurocomputing, 461*, 370–403.
13. He, Y., Zhang, X., & Sun, J. (2017). Channel pruning for accelerating very deep neural networks.
14. Jacob, B., Kligys, S., Chen, B., Zhu, M., Tang, M., Howard, A., Adam, H., & Kalenichenko, D. (2017). Quantization and training of neural networks for efficient integer-arithmetic-only inference.
15. Pfeiffer, K., Rapp, M., Khalili, R., & Henkel, J. (2023). Cocofl: Communication- and computation-aware federated learning via partial NN freezing and quantization. Trans. Mach. Learn. Res. **2023**.
16. Diao, E., Ding, J., & Tarokh, V. (2021). Heterofl: Computation and communication efficient federated learning for heterogeneous clients. In: International Conference on Learning Representations.
17. Tan, M., Chen, B., Pang, R., Vasudevan, V., Sandler, M., Howard, A., & Le, Q. V. (2019). Mnasnet: Platform-aware neural architecture search for mobile. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, pp. 2820–2828.
18. Zhang, S., Du, Z., Zhang, L., Lan, H., Liu, S., Li, L., Guo, Q., Chen, T., & Chen, Y. (2016). Cambricon-x: An accelerator for sparse neural networks. In: 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), pp. 1–12. IEEE
19. Reddy, T. T., Velagaleti, S., Satyanarayana, B., & Kumar, G. P. (2025). Hardware efficient arithmetic reconfigurable fully homomorphic encryption (arfhe) accelerator of low power iot based risc-v processor. *Analog Integrated Circuits and Signal Processing, 124*(1), 1–15.
20. Liu, S., & Karanth, A. (2021). Dynamic voltage and frequency scaling to improve energy-efficiency of hardware accelerators. In: 2021 IEEE 28th International Conference on High Performance Computing, Data, and Analytics (HiPC), pp. 232–241. IEEE
21. Ahmed, M. A., Pfeiffer, K., Khdr, H., Abboud, O., Khalili, R., & Henkel, J. (2025). Accelerated training on low-power edge devices. arXiv preprint arXiv:2502.18323
22. Banbury, C., Reddi, V. J., Torelli, P., Holleman, J., Jeffries, N., Kiraly, C., Montino, P., Kanter, D., Ahmed, S., Pau, D., et al. (2021). Mlperf tiny benchmark. arXiv preprint arXiv:2106.07597.
23. Ignatov, A., Timofte, R., Kulik, A., Yang, S., Wang, K., Baum, F., Wu, M., Xu, L., & Van Gool, L. (2019). Ai benchmark: All about deep learning on smartphones in 2019. In: 2019 IEEE/CVF International Conference on Computer Vision Workshop (ICCVW), pp. 3617–3635. https://doi.org/10.1109/ICCVW.2019.00447
24. Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. Advances in neural information processing systems **25**.
25. Gidaris, S., & Komodakis, N. (2015). Object detection via a multi-region and semantic segmentation-aware cnn model.
26. Lecun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE, 86*(11), 2278–2324. https://doi.org/10.1109/5.726791
27. He, K., Zhang, X., Ren, S., & Sun, J. (2015). Deep residual learning for image recognition. CoRR arXiv:abs/1512.03385.
28. Simonyan, K., & Zisserman, A. (2015). Very deep convolutional networks for large-scale image recognition. arXiv:abs/1409.1556
29. AMD: AMD-Ryzen-Processors. https://www.amd.com/de/support/cpu/amd-ryzen-processors
30. Nvidia: Nvidia RTX 3090. https://www.nvidia.com/de-de/geforce/graphics-cards/30-series/rtx-3090-3090ti/
31. Nvidia: Jetson TX2 NX. https://www.nvidia.com/de-de/autonomous-machines/embedded-systems/jetson-tx2/
32. Nvidia: Jetson Orin Nano. https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-orin/
33. Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al. (2019). Pytorch: An imperative style, high-performance deep learning library. Advances in neural information processing systems **32**.
34. Google: Tensorflow. https://www.tensorflow.org/
35. Meta AI: Sequential Container PyTorch. https://pytorch.org/docs/stable/generated/torch.nn.Sequential.html
36. Bonghi, R. (2019). jetson-stats. https://github.com/rbonghi/jetson_stats. Accessed: 2025-05-27
37. Khdr, H., Oğuz, Y., & Henkel, J. (2024). An open-source tool for analyzing the time efficiency of machine learning on edge devices. In: 2024 International Conference on Microelectronics (ICM), pp. 1–6. IEEE.