SoftwareX 35 (2026) 102865

Contents lists available at ScienceDirect

SoftwareX

journal homepage: www.elsevier.com/locate/softx

FLOPpy: A hardware-agnostic Python library to monitor the computational cost of machine and deep learning algorithms

Francesco Scala [a,*], Francesco Mandarino [b], Liliana Martirano [a], Luigi Pontieri [a]

[a] Institute of High Performance Computing and Networking (ICAR-CNR), Via P. Bucci, 87036 Rende (CS), Italy
[b] Dept. Computer Engineering, Modeling, Electronics, and Systems Engineering (DIMES), University of Calabria, 87036 Rende (CS), Italy

ARTICLE INFO

Keywords:
Green AI
Computational workload
Hardware-agnostic
Deep learning
Machine learning
Floating point operations
Bit-operations

ABSTRACT

The growing awareness of the computational and environmental costs associated with running Machine and Deep Learning models has increased interest in efficiency-aware evaluation. Model performance is commonly assessed through predictive metrics such as accuracy, which do not capture the computational cost required to obtain such results. Existing indicators like execution time, energy consumption, or CO_2 emissions depend strongly on hardware and infrastructure characteristics, limiting comparability and reproducibility. *FLOPpy* addresses this limitation by providing hardware-agnostic estimates of the computational effort required during both training and inference through the number of Floating Point Operations (FLOPs). Beyond this widely adopted metric, *FLOPpy* introduces the estimation of Bit-Operations (BOPs) by dynamically intercepting tensor precisions at runtime. This enables a precision-aware quantification of computation, capturing efficiency aspects that FLOPs alone cannot reflect. By jointly modeling operation count and numerical precision, the proposed library establishes a more faithful and comprehensive characterization of computational cost. *FLOPpy* integrates seamlessly with widely used Machine and Deep Learning libraries, allowing researchers and practitioners to incorporate reproducible, precision-aware efficiency analysis into existing experimental workflows.

Metadata

Code metadata.

Code metadata description	Metadata
Current code version	v0.1.1
Permanent link to code/repository used for this code version	https://github.com/Franco7Scala/FLOPpy
Permanent link to Reproducible Capsule	https://codeocean.com/capsule/4852074
Legal Code License	GNU Public License v3.0 (GPL3)
Code versioning system used	Git
Software code languages, tools, and services used	Python
Compilation requirements, operating environments & dependencies	scikit-learn, torch, psutil, wandb
If available Link to developer documentation/manual	https://floppy.readthedocs.io
Support email for questions	francesco.scala@icar.cnr.it

1. Motivation and significance

The growing computational and environmental impact of Machine and Deep Learning models has motivated efficiency-aware evaluation beyond predictive metrics such as accuracy or error rates. Green AI studies [1–3] emphasize that model assessment should also account for the computational cost of training [4–9] and inference [10–12]. However, commonly used indicators such as execution time, energy consumption, and CO_2 emissions [13–18] strongly depend on hardware and infrastructure, limiting reproducibility and comparability.

A more reproducible and hardware-independent indicator of computational demand is the number of Floating Point Operations (FLOPs), which quantify the arithmetic operations required to execute a model and reflect its intrinsic computational workload (see Appendix A

* Corresponding author.
Email address: francesco.scala@icar.cnr.it (F. Scala).

https://doi.org/10.1016/j.softx.2026.102865
Received 9 April 2026; Received in revised form 25 June 2026; Accepted 4 July 2026
Available online 14 July 2026

for details). Nevertheless, existing FLOPs-based profilers (e.g., `thop` [19], `ptflops` [20], `fvcore`, *torch.profiler*) ignore operand precision and cannot capture the efficiency gains introduced by quantization. Complementary metrics such as Bit-Operations (BOPs) [21], which incorporate operand precision, provide a closer proxy to hardware-level cost (see Appendix B for details). A representative example of how FLOPs-only analyses may misrepresent efficiency in memory-bound settings can be observed in Large Language Models [8,22], where quantization reduces memory-related cost (captured by a decrease in BOPs) while increasing FLOPs due to on-the-fly dequantization required by current hardware. Nevertheless, the use of BOPs is currently confined largely to ad hoc implementations, rather than being integrated into accessible, general-purpose tools. Existing workload estimation methodologies are generally based either on static architectural analysis or hardware-dependent execution measurements. These two perspectives capture complementary aspects of computational cost: static methods estimate theoretical complexity from model architecture, whereas execution measurements reflect device-specific runtime behavior. However, neither provides a hardware-independent characterization of the computation actually executed by a model. As a result, current methods lack a unified framework for hardware-independent, execution-aware, and precision-aware workload estimation.

To address this gap, we introduce *FLOPpy*, a Python library that implements an execution-level computational modeling approach for hardware-independent estimation of computational workload in Machine and Deep Learning models. Rather than inferring complexity solely from architectural descriptions or measuring hardware-dependent execution characteristics, *FLOPpy* estimates workload from the operations effectively executed during model execution. The library integrates with widely used libraries such as Scikit-learn[1] and PyTorch,[2] supporting hardware-independent workload estimation during both training and inference without modifying existing workflows. Conceptually, *FLOPpy* formalizes workload estimation as an execution-level computational modeling problem operating at runtime dispatch level. This perspective bridges the gap between static complexity estimation and hardware-dependent profiling by reconstructing computational demand from the sequence of operations actually dispatched during execution while remaining independent of the underlying hardware platform. By intercepting tensor operations during execution, the framework captures the effective algorithmic complexity of the complete training lifecycle, including forward propagation, backward propagation, optimizer updates, and dynamic execution patterns. Consequently, workload estimation reflects the actual computational graph traversed during execution rather than an architectural abstraction, enabling the analysis of dynamic behaviors such as conditional execution, routing mechanisms, and adaptive computation. By intercepting tensor representations at runtime, *FLOPpy* also identifies their numerical precision and incorporates this information into the estimation process, extending traditional FLOPs-based analyses with precision-aware workload estimation and providing a closer proxy of hardware-level computational effort. The resulting dual characterization, combining operation count and numerical precision, provides a more comprehensive representation of computational demand than FLOPs-only analyses.

Its modular architecture facilitates extensions to additional libraries and model types.

1.1. Experimental setting

In practical usage, *FLOPpy* is integrated into the experimental pipeline by initializing a monitoring environment before training or inference. The user specifies the models to be analyzed together with optional configuration parameters, such as logging and output management.

During execution, the library monitors the operations performed by the selected models and estimates the number of floating point operations associated with both training and inference. By also intercepting tensor data types at runtime, *FLOPpy* can incorporate operand precision into the estimation process, enabling complementary computation of Bit-Operations. This process is transparent and does not require modifications to the model implementation. At the end of execution, *FLOPpy* reports workload statistics combining FLOPs and BOPs, enabling hardware-independent comparison of estimated computational demand across models.

1.2. Related work

Several tools have been proposed to estimate the computational footprint of Machine Learning experiments (see Table 1). Green AI libraries such as CodeCarbon [16] estimate the environmental impact of computational workloads by monitoring execution time, energy consumption, and associated CO_2 emissions. Although these approaches provide useful insights into sustainability, their measurements depend strongly on hardware characteristics and infrastructure-specific factors, which limit reproducibility across different computational environments.

Complementary approaches estimate neural network complexity through FLOPs. Tools such as *ptflops* [20], *thop* [19], *torchinfo*, *fvcore*, *calflops* [23], *Flops Profiler* (by DeepSpeed) and *torch.profiler* estimate FLOPs using analytical layer-based formulas. Because this method relies on the mathematical structure of the model rather than the underlying hardware, it provides a hardware-agnostic approximation of computational workload. However, most existing tools primarily estimate inference complexity and only partially capture training-related operations such as the backward pass, loss evaluation and optimizer updates. Among them, the native *torch.profiler* constitutes the closest alternative to our approach, as it dynamically traces operations at the C++ and CUDA levels rather than relying exclusively on static architectural approximations. Nevertheless, it is primarily designed as a low-level debugging utility, making continuous multi-epoch profiling impractical due to substantial tracing overheads, and its FLOPs counting is restricted to a subset of ATen kernels, leading to systematic underestimation of the full training workload. Moreover, most existing FLOPs estimation libraries are designed for specific Deep Learning frameworks and offer limited support for traditional Machine Learning models. Furthermore, they generally remain agnostic to operand precision and cannot capture the computational impact of quantization techniques.

FLOPpy addresses these limitations by supporting both training and inference across PyTorch and Scikit-learn. Unlike tracing-based profilers, it dynamically intercepts tensor operations without storing execution traces, enabling lightweight precision-aware monitoring through FLOPs and BOPs.

Unlike tracing-based profilers, *FLOPpy* dynamically intercepts tensor shapes without storing execution traces, enabling lightweight end-to-end monitoring with negligible memory overhead. It also extends conventional FLOPs-based profiling by incorporating precision-aware Bit-Operations, enabling the evaluation of quantized models and providing a unified framework for holistic computational reporting in Green AI workflows.

2. Software description

The following section outlines the software architecture, its main functionalities and provides a concise overview of the benchmarking experiments conducted to assess the capabilities, computational overhead, and applicability of *FLOPpy* across heterogeneous Machine Learning and Deep Learning workloads.

2.1. Software architecture

Fig. 1 illustrates the UML class diagram of *FLOPpy*, which is composed of three main components: *(i)* the **Tracker** module, centered on the `FLOPpyTracker` class, which serves as the primary entry point

[1] https://scikit-learn.org/.
[2] https://pytorch.org/.

Table 1
Feature comparison between *FLOPpy* and state-of-the-art profiling libraries.

Library	Frameworks	FLOPs	BOPs	E2E Pipeline*	Classical ML	HW-Agnostic
`fvcore` (Meta)	PyTorch	✓	×	Fwd	×	✓
`thop`	PyTorch	✓	×	Fwd	×	✓
`ptflops`	PyTorch	✓	×	Fwd	×	✓
`calflops`	PyTorch	✓	×	Fwd	×	✓
`torchinfo`	PyTorch	✓	×	Fwd	×	✓
`Flops Profiler`	PyTorch	✓	×	Fwd,Bwd	×	✓
`torch.profiler`	PyTorch	✓	×	Fwd,Bwd	×	×
`CodeCarbon`	Any (System Level)	×	×	N/A (Energy Only)	✓	×
***FLOPpy* (Ours)**	**PyTorch, Scikit-learn**	✓	✓	**Fwd,Bwd,Loss,Opt**	✓	✓

* The end-to-end pipeline indicates whether the library tracks operations beyond the standard model's forward pass (e.g., backward pass, loss, optimizer steps).

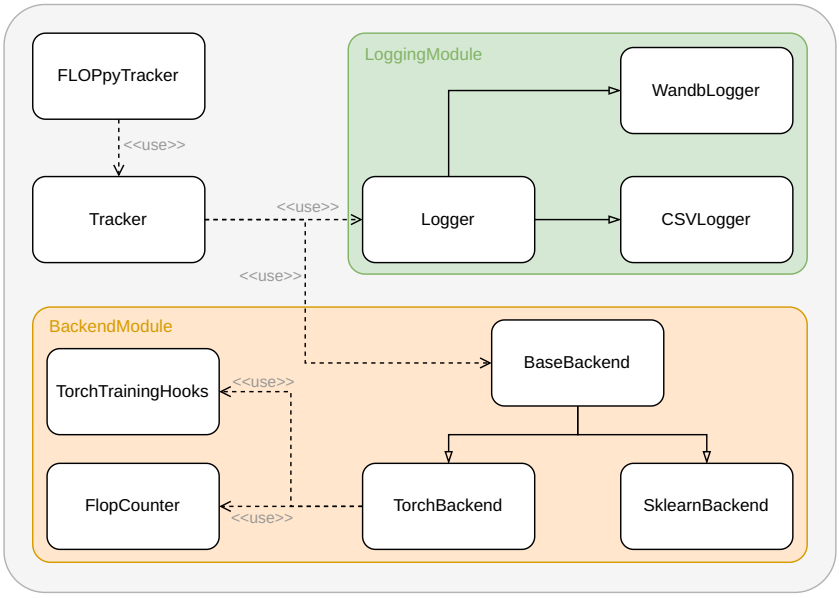

Fig. 1. Architecture of the *FLOPpy* framework.

for users; *(ii)* the **BackendModule**, which employs a provider pattern through `BaseBackend` to support specialized tracking for `Torch` (via `TorchTrainingHooks` and `FlopCounter`) and `Scikit-learn` models; and *(iii)* the **LoggingModule**, which manages experiment tracking through extensible components such as `CSVLogger` and `WandbLogger`.

The software architecture is modular by design. The base `Tracker` class manages state, coordinates tracking routines, and synchronizes data between the backend and the loggers. The `FLOPpyTracker` class provides a high-level *Facade* interface.

Additional logging modules and computational backends can be integrated by extending `BaseBackend`.

The current implementation of *FLOPpy* supports PyTorch and Scikit-learn.

For PyTorch models, *FLOPpy* bypasses traditional `nn.Module` hooks, which miss operations outside module boundaries (e.g., custom losses, residual connections, optimizer updates). Instead, it leverages `TorchDispatchMode` to intercept all C++ ATen operations, ensuring full coverage of the forward pass, backward pass, and parameter updates (see Appendix A for details). To overcome limitations of conventional profilers, *FLOPpy* introduces *Fused-Kernel Transparent Profiling*, enabling consistent accounting of vectorized and fused operations across CPU and GPU. By operating at the dispatch level, it provides a hardware-agnostic and structurally independent estimate of the total computational workload. Minor CPU/GPU differences arise only from backend-specific implementations (e.g., fused versus decomposed operations), reflecting the executed computational graph. Runtime interception of tensor data types simultaneously enables precision-aware BOP estimation.

The dispatch-level tracking further enables support for advanced and dynamic architectures, including Mixture-of-Experts (MoE) models with runtime routing, Hugging Face's models, Graph Neural Networks via PyTorch Geometric (PyG) with potential benefits for applications such as [24,25], recurrent architectures (RNNs), Vision Transformers (ViT), multimodal models, and distributed training. The framework also captures training-time strategies such as gradient accumulation, gradient checkpointing (enabling analysis of memory–compute trade-offs), and parameter-efficient fine-tuning methods (e.g., LoRA), accurately reflecting their impact on computational cost. Additionally, *FLOPpy* supports autoregressive inference workloads by distinguishing between prefill and decode phases (including KV-cache usage), and enables independent tracking of multiple models in alignment pipelines (e.g., DPO).

Supporting Scikit-learn models requires a different approach due to the absence of dynamic computational graphs, tensor dispatchers, or standardized module hierarchies. *FLOPpy* implements a dynamic method-wrapping strategy through the `SklearnBackend`, intercepting standard estimator methods such as `fit()`, `predict()`, `predict_proba()`, and `transform()`. The wrapped methods preserve the original behavior while capturing input and output data dimensions at runtime. Based on the model type, the backend applies analytical complexity formulas to estimate FLOPs, for example using matrix multiplication costs for linear models or distance computations for k-nearest neighbors. This approach enables transparent monitoring of

classical Machine Learning pipelines without requiring modifications to user code.

2.2. Software functionalities

FLOPpy estimates computational workload by analytically counting the mathematical operations associated with model execution during both training and inference. The main functionalities of the software are summarized as follows:

- **Operation-level estimation:** The library estimates computational cost directly from intercepted ATen operations, avoiding reliance on static layer-based approximations;
- **End-to-end workload tracking:** In addition to forward passes, the estimation includes operations related to loss evaluation and optimization steps, ensuring that both training and inference phases are fully accounted for;
- **Precision-aware analysis:** By capturing tensor data types during execution, the library complements FLOPs with Bit-Operations, enabling a more accurate characterization of hardware-level computational effort, particularly for quantized models;
- **Integration and data access:** The library integrates with experiment tracking tools (e.g., *Weights & Biases*) and supports exporting detailed logs and architectural summaries. Computed metrics are accessible programmatically and provided at a fine-grained level, including separate estimates for training and inference as well as aggregated results.

2.3. Software benchmarking and experimental assessment

To assess *FLOPpy*, we conducted a set of benchmarking experiments evaluating: *(i)* agreement with existing FLOPs profiling tools, *(ii)* runtime and memory overhead across heterogeneous architectures, *(iii)* quantization-aware computational analysis through FLOPs and precision-aware BOPs, *(iv)* robustness of the Scikit-learn backend under different estimator configurations, and *(v)* correlation between the estimated hardware-agnostic metrics and physical execution costs.

Table 2
Description of tracker parameters.

Parameter	Description
`model`	The model to be tracked
`optimizer`*	Optimizer used during training, enabling estimation of optimization overhead
`loss_fn`*	Loss function, allowing inclusion of loss computation cost
`tokenizer`*	Tokenizer, allowing inclusion of tokenization cost
`run_name`	Identifier of the experiment
`export_path`	Path for exporting logs and statistics
`wandb_config`	Configuration for logging to Weights & Biases

* For PyTorch and Hugging Face models, this parameter is optional; if omitted, optimizer FLOPs are not computed. For Scikit-learn models, it is ignored.

The experiments include CNNs, Vision Transformers, encoder and decoder Transformers, Mixture-of-Experts models, Graph Neural Networks, recurrent architectures, multimodal models, and classical Machine Learning estimators. Results show that *FLOPpy* captures complete training workloads across heterogeneous architectures. BOPs complement FLOPs for quantized models, and both metrics correlate consistently with execution time and energy consumption.

A detailed description of the experimental setup, benchmarking methodology, and complete quantitative results are reported in Appendix C.

3. Illustrative examples

This section provides illustrative examples of integrating *FLOPpy* into standard machine learning and deep learning pipelines.

The workflow consists of three main steps: model definition, tracker initialization, and execution with result retrieval. Examples of supported models, including PyTorch, Hugging Face, and Scikit-learn, are shown in Listing 1.

The tracking process is initialized by instantiating the `FLOPpyTracker`, which accepts the model and optional parameters

```python
import torch.nn as nn

from floppy import FLOPpyTracker, WandbConfiguration
from transformers import AutoModel
from sklearn.ensemble import RandomForestClassifier


wandb_config = WandbConfiguration(
    project_name="your_project_name",
    group_name="your_group_name",
    reporter_key="your_wandb_key_here"
)

# 1. PyTorch Model
model = nn.Sequential(nn.Linear(10, 10), nn.ReLU())
loss_fn = ALossFunction()
optimizer = AnOptimizer()

# 2. Hugging Face Transformer Model
model = AutoModel.from_pretrained("a_model")
tokenizer = AutoTokenizer.from_pretrained("a_model")
loss_fn = ALossFunction()
optimizer = AnOptimizer()

# 3. Scikit-learn Model
model = RandomForestClassifier(n_estimators=100)
```

Listing 1. Definition of models supported by `FLOPpyTracker`, including PyTorch, Hugging Face, and Scikit-learn examples.

```python
# Initialize and run the tracker
tracker = FLOPpyTracker(run_name="experiment_name").run(
    model=model,
    tokenizer=tokenizer,
    optimizer=optimizer,
    loss_fn=loss_fn,
    export_path="path/to/export/results/result.csv",
    wandb_config=wandb_config
)
```

Listing 2. Initialization of `FLOPpyTracker`. The interface is consistent across model types; parameters such as `optimizer` and `loss_fn` are omitted for Scikit-learn models.

```python
# Do something with the model

# 1. and 2. Neural Model
model = trainer(model, loss_fn, optimizer)
loader = DataLoader(some_data, batch_size=16)
num_epochs = 10

for _ in range(num_epochs):
    for xb, yb in loader:
        optimizer.zero_grad()
        y_hat = model(xb)
        loss = loss_fn(y_hat, yb)
        loss.backward()
        optimizer.step()
        tracker.batch()

    tracker.epoch()

# 3. Scikit-learn Model
model = model.fit(X, Y)
y = model.predict(x)

# Access the comprehensive FlopReport with all statistics
report = tracker.report(print_summary=True)
```

Listing 3. Model execution and retrieval of computational workload statistics.

```
======================================================================
 FLOPpyTracker Summary "experiment_name"
======================================================================
System Environment:
  - System    : Linux (x86_64) | 2016 GB RAM
  - CPU       : AMD EPYC 7742 64-Core Processor | 128 Physical Cores
  - GPU       : 1x NVIDIA A100-SXM4-80GB
  - Python    : 3.10.13
  - Libs      : PyTorch 2.2.2+cu121 | Scikit-learn 1.6.1
Modules tracked details:
  - Device    : CUDA:0
  - Model     : torch.nn.modules.container.Sequential
  - Loss      : CrossEntropyLoss
  - Optimizer : Adam
Computational Workload Breakdown:
  - Model (Forward)         :   414.72 KFLOPs |  13.27 MBOPs
  - Model (Backward)        :   500.22 KFLOPs |  16.01 MBOPs
  - Loss (Forward)          :    11.52 KFLOPs | 368.64 KBOPs
  - Optimizer (Update)      :   136.44 KFLOPs |   4.37 MBOPs
----------------------------------------------------------------------
OVERALL TOTAL FLOPs and BOPs:     1.06 MFLOPs |  34.01 MBOPs
======================================================================
Tracking & Integrations:
  - Export Path: experiment_name.csv
  - W&B Project: experiment_name
  - W&B group  : your_group_name
======================================================================
```

Listing 4. Example of a complete profiling report generated by *FLOPpy* for a PyTorch training experiment.

for controlling logging, exporting, and monitoring (Listing 2). The main configuration parameters are summarized in Table 2.

During execution, the tracker transparently monitors the computational workload of the model in both training and inference phases.

After execution (Listing 3), the user can access a `FLOPpyReport` object containing detailed statistics.

The report includes: (i) a breakdown of FLOPs and BOPs into model (`model_forward_flop/bop` and `model_backward_flop/bop`), optimizer (`optimizer_flop/bop`), and loss (`loss_flop/bop`) components; (ii) optional preprocessing operations (e.g., tokenization for language models), reported as `preproc_ops`; and (iii) metadata describing the execution environment, including system configuration (OS, CPU, memory, GPU) and software/library versions.

To complement aggregate metrics, `tracker.batch()` and `tracker.epoch()` enable incremental logging of computational workload (optional). This functionality supports the analysis of temporal workload trends, with intermediate metrics automatically exported to CSV files and synchronized with Weights & Biases for real-time visualization.

All metrics are programmatically accessible, enabling custom analyses and comparisons across experiments.

4. Impact

FLOPpy provides a unified, hardware-agnostic tool for estimating floating point operations, enabling systematic analyses of the trade-offs between predictive performance and computational cost of ML and DL models. By complementing FLOPs with precision-aware metrics such as Bit-Operations, the library further supports the analysis of efficiency gains introduced by quantization and mixed-precision techniques. In quantized settings, this dual-metric view can expose non-trivial trade-offs, where reduced precision decreases memory-related costs (captured by BOPs) while increasing algorithmic overhead (captured by FLOPs), highlighting limitations of FLOPs-only profiling. This capability can support efficiency-aware model design, compression strategies, and architecture selection workflows where computational constraints must be considered alongside predictive performance.

The library also enhances existing research practices by complementing traditional performance metrics with a standardized, reproducible measure of computational demand, supporting fairer benchmarking across architectures and experimental settings.

By integrating seamlessly into existing pipelines with minimal code changes, *FLOPpy* enables routine monitoring of computational workload, allowing efficiency considerations to guide model development and evaluation. These capabilities are particularly relevant in three practical scenarios: efficiency-aware model optimization, deployment planning under computational constraints, and reproducible benchmarking of dynamic learning architectures.

Support for PyTorch and Scikit-learn enables broad applicability in research and industry, with potential for large-scale use, while the modular architecture facilitates future extensions and allows the library to evolve alongside the rapidly changing Machine and Deep Learning ecosystem. Beyond academia, *FLOPpy* can support deployment and resource-allocation decisions where computational efficiency is critical.

Representative applications include:

Edge deployment and large-scale quantization. In resource-constrained deployment scenarios, such as LLM inference on edge devices, static FLOPs profilers fail to distinguish between full-precision and quantized models. *FLOPpy* enables fine-grained analysis by jointly reporting FLOPs and BOPs, revealing, for example, substantial reductions in BOPs under 4-bit quantization together with additional computational overhead from runtime dequantization. This provides a more accurate basis for selecting deployment strategies in memory- and bandwidth-constrained environments.

Model compression and compute budgeting. In resource-constrained development scenarios, selecting among pruning, quantization, and parameter-efficient fine-tuning strategies requires balancing predictive performance and computational cost. *FLOPpy* enables fine-grained analysis by jointly reporting FLOPs and BOPs, allowing practitioners to quantify the computational impact of alternative optimization strategies. This provides a more accurate basis for model selection under explicit computational budgets and efficiency constraints.

Reproducible efficiency benchmarking. In research settings involving dynamic architectures (e.g., Mixture-of-Experts or adaptive routing models), FLOPs derived from static graphs may not reflect actual execution patterns. *FLOPpy* instead captures the effective computational workload at runtime in a hardware-agnostic manner, enabling reproducible efficiency comparisons across different hardware platforms and ensuring that reported complexity metrics remain consistent across experimental environments.

5. Conclusions

This paper presented *FLOPpy*, a Python library for hardware-independent estimation of computational workload through FLOPs and precision-aware BOPs. By adopting an execution-level computational modeling approach and supporting both PyTorch and Scikit-learn, *FLOPpy* enables reproducible workload characterization with minimal changes to existing ML and DL workflows. The proposed execution-level computational modeling approach provides a unified framework for hardware-independent workload characterization across heterogeneous ML and DL ecosystems, complementing traditional predictive evaluation with reproducible measures of computational demand.

Future work will extend support to additional libraries and model ecosystems, including TensorFlow and JAX, broaden operation coverage, investigate richer cost models incorporating memory and communication costs, and explore efficiency-aware optimization workflows such as model compression and compute-budget-constrained model selection. We also plan a native C++ backend to further reduce profiling overhead.

CRediT authorship contribution statement

Francesco Scala: Writing – review & editing, Writing – original draft, Visualization, Validation, Supervision, Software, Methodology, Formal analysis, Conceptualization. **Francesco Mandarino:** Software, Methodology, Investigation. **Liliana Martirano:** Writing – review & editing, Writing – original draft. **Luigi Pontieri:** Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

This work was partially supported by research project FAIR (PE00000013), funded by the EU under the program NextGeneration EU and by the Programma Nazionale Ricerca, Innovazione e Competitività per la transizione verde e digitale 2021–2027 (PN RIC 2021–2027), co-funded by the European Union – European Regional Development Fund (ERDF), under Directorial Decree (D.D.) No. 307 of 18 March 2025, through the project SINTESI – Sistemi cyber-fisici INTElligenti e SIcuri per la sanità, l'industria e la società (Project Code: MI 1_93098910503_0000962), implemented under Action 1.1.2 ("Sostegno a un numero limitato di filiere strategiche della ricerca" – Support for a limited number of strategic research value chains). For the National Research Council of Italy (CNR), the project is identified by CUP B39H26000050007.

Appendix A. Analytical formulation of computational workload (FLOPs)

This appendix details the analytical formulations implemented in *FLOPpy* to estimate the number of Floating Point Operations (FLOPs) for both deep learning and classical machine learning models.

In the deep learning context, a factor of 2 is introduced where appropriate to account for Multiply-Accumulate (MAC) operations, which consist of one multiplication and one addition.

A.1. Deep learning backend (PyTorch)

To capture the computational workload across the entire execution pipeline, *FLOPpy* intercepts low-level ATen operations via `TorchDispatchMode`. FLOPs are dynamically computed based on the tensor dimensions observed at runtime. Let N_{in} and N_{out} denote the number of elements in the input and output tensors, respectively.

Element-wise operations. Standard element-wise functions (e.g., `add`, `mul`, `relu`, `sigmoid`, `exp`) perform one operation per element. Broadcasting effects are handled by evaluating the maximum between input and output elements:

$$FLOPs_{elementwise} = \max(N_{out}, N_{in}) \tag{A.1}$$

Complex optimizer operations. Composite operations, including fused optimizer steps and complex element-wise updates (e.g., `addcdiv`, `addcmul`) require multiple arithmetic steps (addition, multiplication or division, and accumulation) per element:

$$FLOPs_{complex\_opt} = 3 \cdot \max(N_{out}, N_{in}) \tag{A.2}$$

Linear algebra operations. For standard matrix multiplications (`mm`, `addmm`) between $M \times K$ and a $K \times N$ matrices:

$$FLOPs_{mm} = 2 \cdot M \cdot N \cdot K \tag{A.3}$$

For batched matrix multiplications (`bmm`, `baddbmm`) with batch size B:

$$FLOPs_{bmm} = 2 \cdot B \cdot M \cdot N \cdot K \tag{A.4}$$

Embeddings and normalization layers. Embedding lookups cost one operation per embedding dimension E for every index processed:

$$FLOPs_{embedding} = N_{indices} \cdot E \tag{A.5}$$

Normalization layers (`layer_norm`, `rms_norm`) require 4 operations per element (mean, variance, scale, shift):

$$FLOPs_{norm} = 4 \cdot N_{in} \tag{A.6}$$

Softmax and loss functions. Softmax and Log-Softmax operations evaluate at 3 operations per element (exponentiation, summation, division):

$$FLOPs_{softmax} = 3 \cdot N_{in} \tag{A.7}$$

Standard loss functions (`cross_entropy_loss`, `nll_loss`) operate at 1 operation per element:

$$FLOPs_{loss} = N_{in} \tag{A.8}$$

Convolutions. For intercepted convolutional kernels (`convolution`, `conv`), the cost is derived from the total output elements and the number of parameters per filter, with W_{numel} being the weight tensor size and W_0 being the number of output channels:

$$FLOPs_{conv} = 2 \cdot N_{out} \cdot \frac{W_{numel}}{W_0} \tag{A.9}$$

Scaled dot-product attention (SDPA). For flash attention or standard SDPA operations (`_scaled_dot_product`) with batch size B, number of heads H, sequence lengths L_q and L_k, and head dimension D:

$$FLOPs_{SDPA} = FLOP_{QK^T} + FLOP_{softmax} + FLOP_{AttnV} \tag{A.10}$$

Where $FLOP_{QK^T} = 2 \cdot B \cdot H \cdot L_q \cdot L_k \cdot D$, $FLOP_{softmax} = 3 \cdot B \cdot H \cdot L_q \cdot L_k$, and $FLOP_{AttnV} = 2 \cdot B \cdot H \cdot L_q \cdot L_k \cdot D$.

Graph neural networks (Routing & scatter). Routing operations (`scatter`, `index_add`, `gather`) apply 1 operation per output element:

$$FLOPs_{routing} = N_{out} \tag{A.11}$$

Bin-counting (`bincount`) evaluates 1 operation per input element:

$$FLOPs_{bincount} = N_{in} \tag{A.12}$$

Pooling and upsampling. Standard pooling operations (`max_pool`, `avg_pool`) scale linearly with the output elements:

$$FLOPs_{pool} = N_{out} \tag{A.13}$$

Upsampling operations evaluate at 4 operations per output element (standard bilinear/bicubic interpolation estimations):

$$FLOPs_{upsample} = 4 \cdot N_{out} \tag{A.14}$$

Sorting and mixture of experts (MoE). Sorting algorithms (`sort`, `argsort`, `topk`) scale based on the expected $O(N \log N)$ complexity of the input elements:

$$FLOPs_{sort} = N_{in} \cdot \log_2(\max(2, N_{in})) \tag{A.15}$$

Recurrent neural networks. Intercepted RNN operations (`lstm`, `gru`, `rnn`) are modeled with a baseline approximation of 8 operations per input element across the sequence length, identical across gated and non-gated variants:

$$FLOPs_{rnn} = 8 \cdot N_{in} \tag{A.16}$$

A.2. Classical machine learning backend (Scikit-learn)

For classical machine learning models, FLOP estimation depends on dataset size and model-specific parameters. Let N denote the number of samples, F the number of features, T the number of iterations, and C_{out} the number of output classes.

Linear and generalized linear models. For Ordinary Least Squares (`LinearRegression`), the analytical solution requires matrix operations during the `fit` phase. The coefficients 2 and $\frac{2}{3}$ derive from the standard asymptotic complexity of the matrix multiplication ($X^T X$) and the subsequent matrix decomposition (e.g., Cholesky factorization) required for the exact solution:

$$FLOPs_{OLS\_fit} = 2 \cdot N \cdot F^2 + \frac{2}{3} \cdot F^3 \tag{A.17}$$

For iterative linear models (e.g., `Ridge`, `Lasso`, `LogisticRegression`, `SGDClassifier`), the cost per iteration is computed as:

$$FLOPs_{Iterative\_fit} = T \cdot N \cdot F \cdot \max(C_{out}, 1) \tag{A.18}$$

During inference (`predict`), the cost for all linear models simplifies to vector dot products:

$$FLOPs_{Linear\_predict} = 2 \cdot F \cdot C_{out} \cdot N \tag{A.19}$$

Tree-based models. For `DecisionTree` and `RandomForest` models, let N_{trees} be the number of estimators (1 for a single tree). The training complexity is primarily bounded by sorting and threshold evaluation. The $\log_2(N)$ term models the average depth of a balanced binary tree, representing the expected number of binary routing comparisons required per sample:

$$FLOPs_{Tree\_fit} = N_{trees} \cdot N \cdot F \cdot \log_2(N) \tag{A.20}$$

During inference, let D_{avg} denote the average tree depth:

$$FLOPs_{Tree\_predict} = N_{trees} \cdot N \cdot D_{avg} \tag{A.21}$$

Support vector machines (SVM). For Kernel SVMs (`SVC`, `SVR`), training cost scales quadratically with samples, while prediction depends on the number of support vectors (N_{sv}):

$$FLOPs_{SVM\_fit} = N^2 \cdot F \tag{A.22}$$

$$FLOPs_{SVM\_predict} = 2 \cdot N_{sv} \cdot F \cdot N \tag{A.23}$$

K-nearest neighbors (KNN). Assuming a standard brute-force search, distance computation dominates both phases. Let N_{train} be the number of memorized points during inference:

$$FLOPs_{KNN\_fit} = N \cdot F \tag{A.24}$$

$$FLOPs_{KNN\_predict} = 2 \cdot N_{train} \cdot F \cdot N \tag{A.25}$$

Clustering and dimensionality reduction. For `KMeans` with K clusters, the expectation-maximization iterations cost:

$$FLOPs_{KMeans\_fit} = T \cdot N \cdot F \cdot K \tag{A.26}$$

For Principal Component Analysis (`PCA`) inference (`transform`) mapping to C_{comp} components:

$$FLOPs_{PCA\_transform} = 2 \cdot N \cdot F \cdot C_{comp} \tag{A.27}$$

Appendix B. Precision-aware computational cost estimation (BOPs)

While FLOPs quantify algorithmic complexity, they are agnostic to numerical precision. To capture hardware-level computational effort, *FLOPpy* introduces the estimation of Bit-Operations (BOPs), which scale FLOPs by the effective operand bit-width.

The general formula to compute the hardware computational effort for a given operation is defined as a linear scaling of the algorithmic complexity by the effective bit-width (b_{eff}) of the operands:

$$BOPs = FLOPs \cdot b_{eff} \tag{B.1}$$

The crucial challenge lies in dynamically and accurately determining b_{eff}, especially in the context of modern quantization libraries that often pack sub-byte weights (e.g., 4-bit) into larger standard memory containers (e.g., 8-bit or 32-bit integers). *FLOPpy* adopts framework-specific heuristics to overcome this limitation.

B.1. Deep learning backend (PyTorch)

For neural network models, standard PyTorch `dtype` inspection is insufficient when handling quantized models, where reduced-precision values are often packed into higher-precision containers. To ensure compatibility with widely used quantization frameworks (e.g., `bitsandbytes`, AutoGPTQ, AWQ) without introducing hard dependencies, *FLOPpy* determines the effective bit-width (b_{eff}) through a hierarchical inspection strategy:

1. **Class-name heuristics (Duck-Typing):** The module class name is inspected to identify quantization wrappers. For example, classes containing `Linear4bit` or `Linear8bit` are mapped to $b_{eff} = 4$ and $b_{eff} = 8$, respectively;
2. **Custom attribute inspection:** The backend probes modules and tensors for configuration attributes (e.g., `layer.bits`, `layer.quantize_config.bits`, `tensor.bits`) to explicitly retrieve the quantization precision;
3. **Native dtype mapping:** If no quantization-specific information is available, the backend falls back to the tensor `dtype`, mapping standard types as follows: `float64/int64` → 64, `float32/int32` → 32, `float16/bfloat16` → 16, and `int8/uint8` → 8. Experimental sub-byte types (e.g., `torch.int4`) are mapped accordingly.

If no reliable precision can be inferred, a default of $b_{eff} = 32$ is assumed as standard single-precision floating-point arithmetic.

B.2. Classical machine learning backend (Scikit-learn)

In classical machine learning workflows, computations are primarily performed using NumPy arrays, where packed low-bit representations are uncommon. Consequently, BOP estimation is directly derived from the input data type.

The `SklearnBackend` intercepts input arrays prior to executing wrapped methods (e.g., `fit`, `predict`, `transform`) and evaluates their `numpy.dtype`. The mapping follows the same convention as in the deep learning backend (e.g., `float32` → 32, `int8` → 8). Given that many Scikit-learn algorithms internally promote data to double precision for numerical stability, the fallback value is conservatively set to $b_{eff} = 64$.

Appendix C. Validation and benchmarking

To validate the proposed framework, we conducted experiments addressing four complementary aspects: *(i)* comparison against existing profiling tools, *(ii)* precision-aware FLOPs/BOPs estimation under quantization, *(iii)* runtime and memory overhead analysis across heterogeneous architectures, and *(iv)* validation of the Scikit-learn backend together with correlations between hardware-agnostic metrics and physical execution costs.

C.1. Comparison with existing profilers

Table C.3 shows that existing FLOPs profilers provide estimates consistent with a single forward pass but do not capture the complete training workload. In contrast, *FLOPpy* dynamically tracks forward pass, backward propagation, loss computation, and optimizer updates across multiple epochs. Unlike tracing-based profilers, the proposed approach intercepts tensor metadata without storing execution traces, resulting in negligible additional memory usage during profiling.

To further validate the accuracy of the estimations, we computed the percentage error of each tool against the theoretical end-to-end training workload derived analytically in Eq. (C.1). As shown in Table C.3, *FLOPpy* achieves an extremely low error margin (+2.06%) by correctly capturing the full computational pipeline. Conversely, static profilers exhibit severe underestimation errors (approximately −66%), as they are structurally limited to the forward pass. It is crucial to note that this massive discrepancy is recorded on a single step; accumulating such structural errors over multiple epochs would render static metrics highly unreliable for profiling complete training workloads. Furthermore, `fvcore` registers a disproportionately high error (−83.16%). This occurs because its underlying implementation counts MAC operations without applying the standard ×2 multiplier to yield true FLOPs (reporting ≈ 1.8 instead of 3.6 GFLOPs for the forward pass alone). By resolving these systematic inaccuracies, *FLOPpy* provides a more consistent and theoretically sound evaluation of the actual hardware workload. In this sense, the comparison against static profilers

Table C.3
Comparison against existing profiling tools on ResNet-18. Static profilers estimate only the forward pass, whereas *FLOPpy* tracks the complete training pipeline. FLOPs are expressed in GFLOPs. The error (%) is computed against the theoretical end-to-end training workload of ≈ 10.8 GFLOPs for a single step, highlighting the severe underestimation of forward-only profilers.

Technique	Estimated Cost (GFLOPs)		Mem. Ov. (MB)	Error (%)	Approach
	Single fwd	Multi fwd (10x)			
CodeCarbon	N/A	N/A	+173.17	N/A	Hardware
fvcore (Fwd)	1.819	1.819	+173.17	-83.16%	Static
thop (Fwd)	3.648	3.648	+152.70	-66.22%	Static
ptflops (Fwd)	3.651	3.651	+173.74	-66.19%	Static
torchinfo (Fwd)	3.628	3.628	+152.70	-66.41%	Static
torch.profiler (Fwd)	3.628	36.302	+173.17	-66.41%	Dynamic
FLOPpy (Fwd)	3.632	36.316	+0.00	-66.37%	Dynamic
FLOPpy (Fwd + Bwd+Loss + Opt)	11.023	110.234	+0.00	**+2.06%**	Dynamic

Table C.4
Execution time overhead across heterogeneous architectures and frameworks. The implementations refer to Hugging Face.

Architecture	Model	Time (ms)		
		w/o *FLOPpy*	with *FLOPpy*	Overhead
Vision Transformer	ViT-Base	2511	4300	71.3%
Encoder Transformer	BERT-Base	2522	4150	64.6%
Decoder Transformer	GPT-2	3492	8415	141.0%
CNN	ResNet-18	397	665	67.7%
Multimodal	BLIP-Base	1038	2818	170.7%
MoE	Qwen1.5-MoE	20,385	31,726	55.6%
GNN	Generic-GNN	2530	3889	53.7%
RNN	Generic-RNN	1467	2027	38.2%
Ensemble Tree	Random Forest	14,074	14,119	0.3%

inherently acts as an ablation study of our dispatch-level tracking mechanism, demonstrating the critical information loss that occurs when relying on standard module hooks.

Furthermore, while static profilers require manual extrapolation of single-pass estimates to approximate multi-epoch workloads, *FLOPpy* directly captures the effective end-to-end computational cost of the training pipeline.

C.2. Runtime overhead and architecture coverage

Table C.4 reports the runtime overhead introduced by *FLOPpy* across heterogeneous architectures, including CNNs, Transformers, Mixture-of-Experts, GNNs, RNNs, multimodal models, and classical Machine Learning estimators. The runtime overhead depends on the adopted backend and the operational density of the target architecture. For PyTorch models, operation-level interception through `TorchDispatchMode` introduces additional latency, particularly for autoregressive architectures such as GPT-2, where a large number of tensor operations are executed sequentially. This overhead originates from the interaction between PyTorch's asynchronous C++ backend and the Python interpreter, since intercepting ATen operations requires repeated Python dispatching and synchronization, which becomes particularly costly in operation-dense workloads. In contrast, the Scikit-learn backend relies on analytical complexity estimation through API wrapping rather than tensor interception, resulting in negligible overhead for classical Machine Learning models. These experiments additionally validate the compatibility of *FLOPpy* with advanced architectures and training paradigms beyond simple feed-forward models.

C.3. Quantization-aware computational analysis

Table C.5 validates the precision-aware capabilities of *FLOPpy*. While FLOPs remain approximately constant across floating-point precisions,

Table C.5
Impact of quantization on FLOPs and BOPs for a DistilGPT-2 model. Reduction is computed w.r.t. Float32. FLOPs are expressed in GFLOPs, BOPS in GBOPs.

Precision	FLOPs (GFLOPs)	BOPs (GBOPs)	Reduction BOPs (%)
Float32	4.108	131.443	-
Float16	4.108	65.722	-50.0%
INT8	4.957	58.927	-55.2%
4-bit	4.957	48.735	-62.9%

Table C.6
Sensitivity analysis of the Scikit-learn backend with respect to estimator parameters. FLOPs are expressed in kFLOPs, BOPs in kBOPs.

Model	Configuration	kFLOPs	kBOPs
Random Forest	n_estimators = 10	2.27	145.48
	n_estimators = 100	22.73	1454.82
	max_depth = 2	10.16	650.61
	max_depth = 20	11.36	727.41
SVM	kernel = 'linear'	33.24	2127.36
	kernel = 'rbf'	38.88	2488.32

BOPs decrease proportionally with reduced operand bit-width. The increase in FLOPs observed for INT8 and 4-bit quantization reflects additional runtime operations introduced by quantization and dequantization procedures, which are dynamically captured by the interception mechanism. These results demonstrate that FLOPs alone are insufficient to characterize the effective computational cost of quantized models, motivating the complementary use of BOPs.

C.4. Scikit-learn backend validation

Table C.6 validates the Scikit-learn backend by analyzing the sensitivity of the estimated workload with respect to estimator-specific hyperparameters. Increasing the number of trees in Random Forests produces an approximately linear increase in FLOPs and BOPs, while deeper trees and more complex kernels also increase the computational cost. These results confirm that the proposed backend dynamically adapts workload estimation according to extracted estimator parameters rather than relying on fixed generic complexity formulas.

Unlike the deterministic computational graphs of neural networks, classical machine learning algorithms exhibit data-dependent execution paths. For example, iterative solvers (e.g., SVMs, Logistic Regression) depend on convergence tolerance, and tree-based models rely on dynamic splitting criteria. Consequently, *FLOPpy*'s analytical approximations provide expected asymptotic bounds rather than cycle-accurate instruction counts. This approach inherently introduces an estimation bias that favors theoretical worst-case or average-case complexities, but it remains essential for providing a consistent hardware-agnostic evaluation.

Table C.7
Correlation between hardware-agnostic workload metrics and physical execution costs across the ResNet family. FLOPs are expressed in GFLOPs, BOPs in TBOPs.

Model	FLOPs (GFLOPs)	BOPs (TBOPs)	Elapsed time (ms)	Energy consumption (mWh)
ResNet-18	348.7	11.16	14.3	3.8
ResNet-34	704.1	22.53	22.6	4.6
ResNet-50	786.8	25.17	36.6	6.5

C.5. Correlation with physical execution metrics and hardware discrepancies

Table C.7 compares the workload metrics estimated by *FLOPpy* against elapsed execution time and energy consumption measured through CodeCarbon. The results show a consistent monotonic relationship between the estimated workload and the observed physical execution costs. Larger architectures exhibit higher FLOPs/BOPs together with increased runtime and energy consumption. To reduce measurement variability and avoid unreliable hardware-level observations caused by background processes and resource contention, all experiments were executed on an otherwise idle machine under controlled conditions. Furthermore, the dynamically intercepted FLOPs closely match the expected theoretical complexity of the architectures.

For instance, a standard ResNet-18 forward pass theoretically requires approximately 1.8×10^9 MACs (Multiply-Accumulate operations), which is equivalent to $FLOPs_{\text{fwd}} \approx 3.6$ GFLOPs per image. Accounting for the full end-to-end training pipeline (where the backward pass and optimizer updates typically demand roughly twice the computational effort of the forward pass, as detailed in [26]) and scaling by a batch size $B = 32$, the expected theoretical workload per training step can be formalized as:

$$\begin{aligned} FLOPs_{\text{step}} &\approx (FLOPs_{\text{fwd}} + FLOPs_{\text{bwd\_opt}}) \times B \\ &\approx (FLOPs_{\text{fwd}} + 2 \cdot FLOPs_{\text{fwd}}) \times 32 \\ &\approx (3.6 \text{ GFLOPs} \times 3) \times 32 \\ &\approx 10.8 \text{ GFLOPs} \times 32 \approx 345.6 \text{ GFLOPs} \end{aligned} \tag{C.1}$$

This analytical baseline of 345.6 GFLOPs tightly matches the 348.7 GFLOPs dynamically captured by *FLOPpy*.

While algorithmic workload and physical costs show a consistent monotonic relationship, physical execution is heavily influenced by hardware bottlenecks like memory bandwidth and kernel overheads. This non-linear dynamic is evident when transitioning from ResNet-34 to ResNet-50: algorithmic FLOPs increase by only 11.7%, but physical execution time jumps by 61% due to memory-intensive 1×1 bottleneck convolutions. This confirms that analytical metrics establish an algorithmic baseline rather than a cycle-accurate hardware simulation. Although *FLOPpy* intentionally abstracts these physical constraints to isolate pure mathematical effort, it actively captures framework-level optimizations. By intercepting ATen operations via `TorchDispatchMode`, it accounts for fused kernels (e.g., `aten._scaled_dot_product`) and mixed-precision bit-widths dynamically. Ultimately, *FLOPpy* provides a rigorous, hardware-agnostic quantification of algorithmic intent that strictly complements physical profiling.

References

[1] Schwartz R, Dodge J, Smith NA, Etzioni O. Green AI. Commun ACM 2020;63:54–63.
[2] Verdecchia R, Sallou J, Cruz L. A systematic review of green AI. Wires Data Min Knowl Discov 2023;13(4):e1507. https://doi.org/10.1002/widm.1507
[3] Barbierato E, Gatti A. Toward green AI: a methodological survey of the scientific literature. IEEE Access 2024;12:23989–4013. https://doi.org/10.1109/ACCESS.2024.3360705
[4] Scala F, Flesca S, Pontieri L. Data filtering for a sustainable model training. In: Proceedings of the 32nd symposium of advanced database systems, villasimius, Italy, June 23rd to 26th, 2024, Vol. 3741 of CEUR Workshop Proceedings. CEUR-WS.org; 2024. p. 205–16. https://ceur-ws.org/Vol-3741/paper26.pdf.
[5] Scala F, Flesca S, Pontieri L. Play it straight: an intelligent data pruning technique for Green-AI. In: Pedreschi D, Monreale A, Guidotti R, Pellungrini R, Naretto F, editors. Discovery science. Cham: Springer Nature Switzerland; 2025. p. 69–85.
[6] Scala F, Flesca S, Pontieri L. An efficient model training framework for green AI. Mach Learn 2025;114(12). https://doi.org/10.1007/s10994-025-06907-w. https://www.scopus.com/inward/record.uri?eid=2-s2.0-105021624837&doi=10.1007%s10994-025-06907-w&partnerID=40&md5=3b946e919fdb21363b66e6f894c9dc95.
[7] Justus D, Brennan J, Bonner S, McGough AS. Predicting the computational cost of deep learning models. In: 2018 IEEE international conference on big data (big data). Los Alamitos, CA, USA: IEEE Computer Society; 2018. p. 3873–82. https://doi.org/10.1109/BigData.2018.8622396
[8] Hoffmann J, Borgeaud S, Mensch A, Buchatskaya E, Cai T, Rutherford E, de Las Casas D, Hendricks LA, Welbl J, Clark A, Hennigan T, Noland E, Millican K, van den Driessche G, Damoc B, Guy A, Osindero S, Simonyan K, Elsen E, Rae JW, Vinyals O, Sifre L. Training compute-optimal large language models, arxiv preprint, arxiv:2203.15556. 2022.
[9] Luccioni AS, Viguier S, Ligozat A-L. Estimating the carbon footprint of BLOOM, a 176B parameter language model. J Mach Learn Res 2023;24:253:1–253:15. http://dblp.uni-trier.de/db/journals/jmlr/jmlr24.html#LuccioniVL23.
[10] Han S, Mao H, Dally WJ. Deep compression: compressing deep neural network with pruning, trained quantization and huffman coding. In: Bengio Y, LeCun Y, editors. 4th international conference on learning representations, ICLR 2016, San Juan, Puerto Rico, may 2–4, 2016, conference track proceedings; 2016. 1510.00149. http://arxiv.org/abs/1510.00149.
[11] Li H, Kadav A, Durdanovic I, Samet H, Graf HP. Pruning filters for efficient ConvNets, arxiv preprint, arxiv:1608.08710. 2017.
[12] Hinton GE, Vinyals O, Dean J. Distilling the knowledge in a neural network. arxiv:1503.02531 [arxiv preprint]. 2015. http://dblp.uni-trier.de/db/journals/corr/corr1503.html#HintonVD15.
[13] Giannini F, Franzè G, Pupo F, Fortino G. A sustainable multi-agent routing algorithm for vehicle platoons in urban networks. IEEE Trans Intell Transp Syst 2023;24(12):14830–40. https://doi.org/10.1109/TITS.2023.3305463
[14] Famularo D, Giannini F, Fortino G, Franzè G. A distributed control architecture for sustainable routing decisions of autonomous vehicle platoons subject to cyber attacks. In: 10th international conference on control, decision and information technologies, CoDIT 2024, vallette, Malta, July 1–4, 2024. IEEE; 2024. p. 1147–52. https://doi.org/10.1109/CoDIT62066.2024.10708430
[15] Strubell E, Ganesh A, McCallum A. Energy and policy considerations for deep learning in NLP, arXiv preprint, arXiv:1906.02243. 2019.
[16] Courty B, Schmidt V, Luccioni S, MarionCoutarel G-K, Feld B, Lecourt J, LiamConnell, Saboni A, supatomic I, Léval M, Blanche L, Cruveiller A, ouminasara, Zhao F, Joshi A, Bogroff A, de Lavoreille H, Laskaris N, Abati E, Blank D, Wang Z, Catovic A, Alencon M, Stechly M, Bauer C. Lucas-otavio, JPW, MinervaBooks, mlco2/codecarbon: v2.4.1; May 2024. https://doi.org/10.5281/zenodo.11171501
[17] Anthony LFW, Kanding B, Selvan R. Carbontracker: tracking and predicting the carbon footprint of training deep learning models, ICML workshop on challenges in deploying and monitoring machine learning systems, arxiv preprint, arXiv:2007.03051. Jul 2020.
[18] Budennyy SA, Lazarev VD, Zakharenko NN, Korovin AN, Plosskaya OA, Dimitrov DV, Akhripkin VS, Pavlov IV, Oseledets IV, Barsola IS, Egorov IV, Kosterina AA, Zhukov LE. eco2AI: carbon emissions tracking of machine learning models as the first step towards sustainable AI. Dokl. Math. 2022;106(1):118–28. https://doi.org/10.1134/S1064562422060230
[19] Ultralytics. THOP: PyTorch-OpCounter; 2019. https://github.com/ultralytics/thop.
[20] Sovrasov V. ptflops: a flops counting tool for neural networks in PyTorch framework; 2018-2024. https://github.com/sovrasov/flops-counter.pytorch.
[21] Baskin C, Liss N, Schwartz E, Zheltonozhskii E, Giryes R, Bronstein AM, Mendelson A. Uniq: uniform noise injection for non-uniform quantization of neural networks. ACM Trans. Comput. Syst. (TOCS) 2021;37(1–4):1–15.
[22] Yang L, Hou Q, Zhu X, Lu Y, Xu LD. Potential of large language models in blockchain-based supply chain finance. Enterp Inf Syst 2025;19(11):2541199. https://doi.org/10.1080/17517575.2025.2541199
[23] ye X. calflops: a FLOPs and params calculate tool for neural networks in PyTorch framework; 2023. https://github.com/MrYxJ/calculate-flops.pytorch.
[24] Martirano L, Scala F, Comito C, Pontieri L. Who drives misinformation? Key node detection with heterogeneous graph neural networks. In: International conference on discovery science. Springer; 2025. p. 47–62.
[25] Gullo F, Mandaglio D, Tagarelli A. Neural discovery of balance-aware polarized communities. Mach Learn 2024;113(9):6611–44. https://doi.org/10.1007/s10994-024-06581-4
[26] Kaplan J, McCandlish S, Henighan T, Brown TB, Chess B, Child R, Gray S, Radford A, Wu J, Amodei D. Scaling laws for neural language models. arxiv:2001.08361. 2020.