Enhancing Energy-Awareness in Deep Learning through Fine-Grained Energy Measurement

SAURABHSINGH RAJPUT, Dalhousie University, Canada
TIM WIDMAYER, University College London, UK
ZIYUAN SHANG, Nanyang Technological University, Singapore
MARIA KECHAGIA, University College London, UK
FEDERICA SARRO, University College London, UK
TUSHAR SHARMA, Dalhousie University, Canada

With the increasing usage, scale, and complexity of Deep Learning (DL) models, their rapidly growing energy consumption has become a critical concern. Promoting green development and energy awareness at different granularities is the need of the hour to limit carbon emissions of DL systems. However, the lack of standard and repeatable tools to accurately measure and optimize energy consumption at fine granularity (*e.g.,* at the API level) hinders progress in this area.

This paper introduces **FECoM (Fine-grained Energy Consumption Meter)**, a framework for fine-grained DL energy consumption measurement. FECoM enables researchers and developers to profile DL APIs from energy perspective. FECoM addresses the challenges of fine-grained energy measurement using static instrumentation while considering factors such as computational load and temperature stability. We assess FECoM's capability for fine-grained energy measurement for one of the most popular open-source DL frameworks, namely TensorFlow. Using FECoM, we also investigate the impact of parameter size and execution time on energy consumption, enriching our understanding of TensorFlow APIs' energy profiles. Furthermore, we elaborate on the considerations and challenges while designing and implementing a fine-grained energy measurement tool. This work will facilitate further advances in DL energy measurement and the development of energy-aware practices for DL systems.

Additional Key Words and Phrases: Energy measurement, Green Artificial Intelligence, Fine-grained energy measurement

1 INTRODUCTION

Deep Learning (DL)-based solutions are employed in an increasing number of areas concerning our day-to-day life, such as, in medicine [7, 59, 63], transportation [37, 86, 89], education [58, 64, 87], and finance [56, 62]. However, the increasing use of DL requires ample computational resources, resulting in an alarming surge in energy consumption. The extensive use of computational resources causes significantly increased CO_2 emissions and financial costs [29]. For instance, training a MegatronLM model [76] consumes enough energy to power three American households for a year [48]. This unsustainable trend is continuing; the computational resources required to train a best-in-class ML model is doubling every 3.4 months [2].

For this reason, it is essential to make software development, specifically using DL, *energy-aware*, *i.e.,* develop DL code optimized from the energy consumption perspective, without compromising models' accuracy [70]. To achieve this goal, we propose to measure energy consumption for DL applications at fine granularity (such as at the API level) and identify energy-hungry API calls; so that we can suggest alternative *energy-efficient* software versions [29].

Authors' addresses: Saurabhsingh Rajput, Dalhousie University, Canada, saurabh@dal.ca; Tim Widmayer, University College London, UK, tim.widmayer.20@ucl.ac.uk; Ziyuan Shang, Nanyang Technological University, Singapore, zshang001@e.ntu.edu.sg; Maria Kechagia, University College London, UK, m.kechagia@ucl.ac.uk; Federica Sarro, University College London, UK, f.sarro@ucl.ac.uk; Tushar Sharma, Dalhousie University, Canada, tushar@dal.ca.

Software engineering researchers have studied the energy footprints of software programs in recent years [17, 30, 32, 34]. Broadly, energy measurement techniques are classified into *hardware-based* and *software-based*. *Hardware-based* techniques use physical devices such as a power monitor to measure the power consumed by a machine at a given time. However, hardware-based techniques are considered very difficult in use [15], because of *syncing* issues, *i.e.,* to sync the start and end times of the program execution with a hardware device, automatically. *Software-based* techniques measure energy by using a set of special-purpose registers, referred to as performance counters (PMCs), in modern processors. These registers count specific hardware events [45], including power consumed by hardware components, such as CPU and memory. Most research studies [1, 32, 44] use software tools such as `perf` [53] and `PowerStat` [11] that use PMCs under the hood to measure energy consumption.

Energy can be measured at different granularities, ranging from coarse-grained system-level to fine-grained API[1] level. System-level measurement considers the overall energy consumption of the entire machine or computing hardware. A program- or a process-level profiling examines the energy used by a software application. Function-level measurement profiles the energy usage within specific code blocks and methods. API-level measurement focuses on the energy footprint of external frameworks called by the software in the form of API call statements. API-level measurement offers the finest granularity in attributing energy consumption to specific code entities.

Despite efforts to improve energy-efficiency of source code, we observe many gaps and deficiencies in the literature [29]. Georgiou *et al.* [32] revealed that the *documentation* of even the most popular DL frameworks, namely TensorFlow and PyTorch, lack an energy-consumption profile of their APIs. Such an energy consumption profile could motivate software developers to explore alternative solutions and make software development more energy-efficient [29]. The primary reason for the lack of energy-aware documentation for DL frameworks is the absence of fine-grained energy consumption measurement techniques [30]. In fact, the majority of the existing approaches allow us to measure energy consumption only at the system-level due to the support offered by hardware and operating system vendors [15].

There have been some efforts to measure energy at a fine-grained level; however, existing approaches for measuring energy consumption, in general, have several deficiencies. For example, existing approaches [61, 91] operate at a coarser granularity, measuring entire functions and cannot be used to measure the energy consumed by at the statement-level, including calls to external frameworks, libraries, and APIs. Furthermore, they support only specific programming languages such as Java, C/C++, and Fortran for CPU architectures, without considering GPU architectures used in most deep learning deployments [16].

Software tool vendors have developed relevant tools, such as *CodeCarbon* [12] and *Experiment Impact Tracker* [38], to estimate power consumption and carbon emissions during the training of DL models. However, these tools focus on the ML *program-level* granularity, leading to sampling intervals exceeding 10 seconds, which is not suitable for fine-grained energy measurements (because the measured source code entity can complete its execution in a fraction of seconds). Moreover, they, including other academic studies so far, overlook the overhead introduced by background processes and temperature fluctuations within the computing environment, resulting in noisy measurements. Bannour *et al.* [5] have shown that existing tools consistently under-report energy consumption and carbon emissions, making them less sensitive to measuring energy at a smaller scale and, hence, making them unsuitable for measuring energy consumption at a finer granularity.

[1] An API (*i.e.,* Application Programming Interface) refers to publicly available elements (*e.g.,* interfaces, classes, methods) in a library or a framework. An API comes with its public reference documentation that explains to the user how a method should be used, properly. Client applications, such as ML-based programs, call these APIs from the DL frameworks *e.g.,* TensorFlow and PyTorch, to implement their functionalities.

At present, to the best of our knowledge, there is no convenient (*i.e.,* easily usable), generic (*i.e.,* that can be applied on various kinds of programs and granularity), and automated noise-free solution for measuring the energy consumption of custom deep learning code at a fine-grained level, such as at the API granularity.

This study *aims* to address the challenge of measuring energy consumption at the API granularity as a crucial step towards using such a mechanism. To this end, we devise a framework *viz.* FECoM, which measures the energy consumed by APIs within a DL framework. FECoM generates an Abstract Syntax Tree (AST) of the input program, applies static instrumentation by using our patching mechanism, and measures the energy consumption of the desired APIs. We empirically investigate how the size of the parameters of an API call affects energy consumption and execution time (RQ2). This empirical analysis provides valuable insights into how data size influences energy consumption, enriching our understanding of the used APIs' energy profile in the context of DL applications. Though we use the developed method to measure energy consumed by a DL framework APIs, it can be used to measure energy consumption at a code block or even statement-level. Given the absence of any standard for DL API energy-consumption at a fine-grained level of measurement, we take a step further and provide a detailed account of challenges and considerations that are vital for researchers designing energy measurement tools (RQ3). Such considerations will facilitate the development of more tools and techniques for energy measurement in the field. Our study makes the following contributions:

- **Method and framework.** Implementing a generic method and framework, FECoM [71], to accurately measure energy consumption at a fine-grained level. Such a method has been instantiated for TensorFlow, to show its feasibility in practice. FECoM enables profiling of TensorFlow APIs, offering benefits such as comparing energy efficiency of different configurations, identifying optimization opportunities, and promoting energy-aware coding practices.
- **Static instrumentation tool.** Developing a static instrumentation tool, *viz. Patcher* [24], enabling necessary program patching for the energy measurement framework.
- **Empirical study.** Conducting an empirical study to evaluate and understand the impact of parameter size on energy consumption for DL APIs, as well as systematically examine the execution reports produced by our FECoM to understand the reasons of failures to measure energy consumption at a fine-gained granularity.
- **Dataset.** Creating a dataset [22] comprising energy profiling data at the API granularity for 528 TensorFlow API calls, covering 44% of TensorFlow APIs. This dataset covers a diverse range of domains and includes API calls with varying input parameter sizes.
- **Documenting the challenges and their mitigation strategies.** Collecting and categorizing challenges that may arise during the development of an energy measurement tool at fine-grained granularity to facilitate the development of effective energy measurement tools and provide insights to researchers and developers in this field. The study also provides a set of mitigation strategies applicable for each of the identified challenges by systematically mining Stack Overflow posts.
- **Guiding the selection of energy measurement techniques.** Proposing a set of criteria to guide researchers and developers in selecting the most suitable energy measurement technique for their specific needs, considering factors such as measurement granularity, sampling rate, language and framework compatibility, hardware support, stability, and automation.

We make our framework FECoM, the dataset, as well as the patching program used for static instrumentation publicly available [20].

2 RELATED WORK

Measuring energy consumption of software systems. A significant number of studies [36, 52, 68, 69] uses physical power meters, such as the Monsoon high voltage power monitor [40], to measure a system's energy consumption. These devices physically measure the electrical power consumed by a given software system. The key benefits of using a hardware power meter are its accuracy and precision. Another way to measure energy consumption is to use software tools [9, 33, 78, 79]. Recent Intel and AMD processors provide the *Running Average Power Limit* (RAPL) interface [90], which can measure the power consumption of a processor at regular intervals through built-in performance counters. RAPL can measure the power consumption up to intervals of approximately 1 ms, translating to a sampling frequency of 1 kHz [42]. Many tools have been built on top of RAPL, such as the Intel Power Gadget (it has been discontinued from usage) [41], PowerTOP [84] and Perf [53]. Power modeling is another method that is used to obtain insights into the energy consumption of software systems [36]. Power modeling techniques estimate energy consumption by considering factors such as the energy characteristics of the hardware and run-time information. However, power modeling techniques' full potential has yet to be unlocked.

Table 1 presents a comprehensive comparison of existing energy measurement techniques, highlighting their key features and limitations. The comparison features were carefully selected to demonstrate the need for a more advanced and comprehensive approach to energy measurement in DL frameworks.

- ***Granularity.*** It determines the level of detail at which energy consumption can be analyzed. While some approaches such as CodeCarbon and FPowerTool provide function-level granularity, others such as JavaIO and Perf operate at the system level. To effectively optimize energy consumption in DL frameworks, it is essential to have a fine-grained approach that can measure energy consumption at the API level.
- ***Sampling rate.*** It determines the frequency at which energy measurements are taken. A higher sampling rate allows for more frequent and detailed energy consumption analysis but can lead to inaccurate measurement due to overheads if not carefully chosen. Most existing approaches have sampling rates ranging from fractions of milliseconds to seconds.
- ***Language and architecture support.*** They are also critical considerations when measuring energy consumption in DL frameworks. Python is the most widely used language for DL, and support for CPU, GPU and RAM architectures is essential for comprehensive energy analysis. While some approaches, such as CodeCarbon and Experiment Impact Tracker, support Python and multiple architectures, others are limited in language and architecture support.
- ***Stability check.*** The ability to ensure power and temperature stability during energy measurement is crucial for obtaining accurate and reliable results. Monsoon, a hardware-based approach, provides both power and temperature stability but lacks the flexibility and ease of use offered by software-based approaches.
- ***Automation.*** The automation of energy measurement is a key feature that can greatly simplify the process of analyzing and optimizing energy consumption in DL frameworks. Most existing approaches require manual instrumentation of the code, which can be time-consuming and error-prone.

Optimizing energy consumption of ML tasks. Previous studies [3, 32, 39, 94] have investigated the energy consumption of different DL models. Algorithmic optimization is a major approach that contributes to reducing the energy consumption of DL models. These optimizations often take place in model pruning, which refers to removing unnecessary connections within a neural network [35, 93], and quantization, which reduces the precision of the weights and activations within a neural network by making them quantized and reducing the memory required for a given

Table 1. Comparison of energy measurement techniques

Approach	Granularity	Sampling Rate	Languages	Architecture	DL Frameworks	Power stability	Temperature stability	Automated
CodeCarbon [12]	Program/Function	15s	Python	CPU(Deprecated), GPU and RAM	✓	×	×	×
FPowerTool [91]	Function	1ms	Fortran/C/C++	CPU	×	×	×	×
JavaIO [61]	System	1ms	Java	CPU	×	×	×	×
Perf [53]	System	1ms	n/a	CPU	×	×	×	×
PowerTOP [84]	System	20s	n/a	CPU and GPU	×	×	×	×
Experiment Impact Tracker [38]	Program/Function	1s	Python	CPU and GPU	✓	×	×	×
Monsoon [40]	Hardware	0.2ms	C#/C++	Machine-level	×	✓	✓	×
FECoM [71]	**API/Program**	**500ms**	**Python**	**CPU, GPU and RAM**	✓	✓	✓	✓

model. For instance, Han et al. [35] found that the energy consumption of deep neural networks can be reduced by using a combined technique consisting of model pruning, weight quantization, and Huffman coding. This approach can reduce the size and computational complexity of the deep learning models without significant degradation to their performance.

Measuring energy consumption at different granularities. Available hardware and software tools measure energy consumption at the system level. To improve the granularity of analysis and recommendations, researchers have attempted to measure the energy consumption at a more fine-grained level, such as at the *process* level [49, 61] at best. Bree et al. [8] attempted method replacement for measuring the energy consumed by the visitor pattern; however, their approach is unsuitable for measuring any given method's energy consumption due to the lack of a generalizable solution. FPowerTool [91] operates at a coarser granularity of function blocks for programming languages written in C, C++, and Fortran in non-dl programs. However, its biggest drawback is that it uses a dynamic instrumentation technique that involves injecting tracing code during the runtime, which would lead to noisy and inaccurate energy measurement data at finer granularities such as API level. Similarly, CodeCarbon [12] and Experiment Impact Tracker [38] are open-source software tools designed to monitor and reduce the CO_2 emissions associated with computing processes, particularly those involved in DL applications. These tools integrate themselves into Python codebases and enable the tracking of emissions based on power consumption and location-dependent carbon intensity. While CodeCarbon and Experiment Impact Tracker can measure power consumption, and hence CO_2 emissions at a function-level granularity through the use of decorators, they have a few drawbacks. Firstly, the process of modifying source code to measure the power consumption for desired methods is manual in nature. For instance, if users of CodeCarbon wish to measure the power consumption of TensorFlow API calls, they have to insert the addition lines-of-code calling the CodeCarbon tool themselves, manually for each TensorFlow API instance. Secondly, and more importantly, it overlooks the overheads introduced by background processes and temperature fluctuations while having a sampling rate of 15 seconds, leading to significant noise in the measured energy consumption. Therefore, to the best of our knowledge, the literature does not offer any approach that measures energy consumption at the method level for DL frameworks.

Gaps in existing research. The existing literature as shown in Table 1 has two critical limitations. First, existing approaches can measure energy consumption at only the system level due to support offered by hardware and operating system vendors. There have been efforts to measure energy consumption at the finer granularity (such as at the function level); however, as discussed, they lack various requirements such as low sampling rate, stability check, support for DL framework and languages such as TensorFlow and python, as well as support for CPU, GPU, and RAM architectures. Second, current approaches require manually instrumenting the code to measure energy consumption. For a software engineer to improve the energy efficiency of a given code conveniently and efficiently, the engineer must have access to an approach that can automate

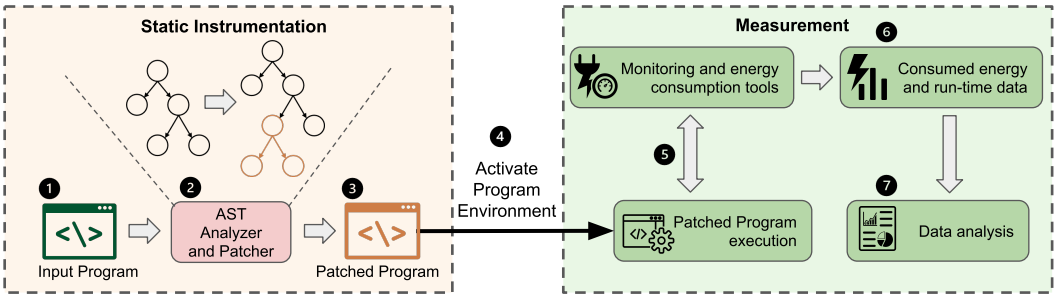


Fig. 1. Architecture of the FECoM framework for energy measurement of individual methods.

the energy measurement process to work at the fine-grained level so that the engineer can take corrective actions, if needed, early. Our proposed approach addresses the gap by providing an automated and generic mechanism to measure energy consumption at the method granularity.

3 APPROACH

This section describes the architecture of the Fine-grained Energy Consumption Meter (FECoM) framework and static instrumentation we devised for fine-grained energy-consumption measurement.

3.1 FECoM Architecture

Figure 1 presents the architecture of the proposed framework, FECoM. FECoM employs static instrumentation to measure fine-grained energy consumption. For a given program, FECoM identifies a set of target API calls and instruments the code around the identified calls. The instrumented code triggers FECoM's measurement module, enabling us to isolate the API from the rest of the program during execution and measure its energy consumption. The measurement module plays a crucial role in ensuring that the temperature and energy consumption remain stable. It performs the necessary checks to verify these conditions. Once the temperature and energy stability criteria are met, the measurement module proceeds to execute the API call specified, capturing the corresponding energy data. In the following, we describe each of FECoM's components in detail [20].

3.2 Static instrumentation

We developed a tool, referred to as *Patcher*, to instrument the code to enable energy-consumption measurement. *Patcher* operates at both the method and project levels, offering fine-grained control over energy profiling. In the context of this work, "method" refers to an API call within the code of a DL project, while "project" encompasses the entire codebase of a DL project hosted in a repository.

Patcher generates an Abstract Syntax Tree (AST) of the input Python script and identifies the libraries and their aliases used within them. This information allows *Patcher* to locate the API calls corresponding to the specified libraries. *Patcher* allows specifying the library for the analysis. For instance, if the user wants to measure the energy consumption of the API's provided by TensorFlow, they specify the name of the library ("TensorFlow" in this case) as an input parameter, and *Patcher* automatically identifies all the TensorFlow API calls. Additionally, *Patcher* identifies class definitions that utilize the required libraries as base classes and keeps track of objects created from these classes. This enables the identification of method calls made through these objects, which are the target calls for energy measurement.

The instrumentation process inserts two source code statements around the target function calls, as shown in Listing 1, which act as breakpoints. The first statement, `before_execution_INSERTED_INTO_SCRIPT`, is placed before the original function call; the statement ensures that the machine has reached a stable state and captures the start time of method execution. The second statement, `after_execution_INSERTED_INTO_SCRIPT`, is inserted after the function call; this statement records relevant information such as the total execution time and energy consumed. We provide the list of used parameters and corresponding descriptions in Table 2.

The Project-level script *Patcher* follows a similar approach as the method-level *Patcher* by inserting the same source code statements before and after the entire script, enabling comprehensive energy-consumption measurement throughout the project's execution.

```
train_notes = np.stack([all_notes[key] for key in key_order], axis=1)
start_times_INSERTED_INTO_SCRIPT =
before_execution_INSERTED_INTO_SCRIPT(
    experiment_file_path=EXPERIMENT_FILE_PATH,
    function_to_run="tensorflow.keras.Input()"
)
inputs = tf.keras.Input(input_shape) #Original API
after_execution_INSERTED_INTO_SCRIPT(
    start_times=start_times_INSERTED_INTO_SCRIPT,
    experiment_file_path=EXPERIMENT_FILE_PATH,
    function_to_run="tensorflow.keras.Input()",
    method_object=None,
    function_args=[input_shape],
    function_kwargs=None)
all_notes = pd.concat(all_notes)
n_notes = len(all_notes)
```

Listing 1. Sample patched code snippet.

Table 2. Description of the arguments or inserted function in the patched code.

Argument/Function	Description
`start_times_INSERTED_INTO_SCRIPT`	Start times determined by before_execution
`before_execution_INSERTED_INTO_SCRIPT`	Patched function added before the original API call
`EXPERIMENT_FILE_PATH`	Path to store experiment data
`function_to_run`	API signature for analysis
`after_execution_INSERTED_INTO_SCRIPT`	Patched function added after the original API call
`method_object`	Object in case of a method call *e.g.*, model in model.compile()
`function_args`	Arguments of the API call
`function_kwargs`	Keyword arguments of the API call

3.2.1 Validation for the static instrumentation tool. To validate *Patcher*, we drew inspiration from Automated Program Repair (APR) techniques [88], adapting them to our use case. The validation process consists of the following steps:

Executability: In the first step, we ensure the executability of the generated patches, confirming that they execute without any syntactical errors. As Python is an interpreted language, we utilize

the `Pylance` language server [57] to validate the patched code for any syntactic or type errors, ensuring a smooth compilation process.

Automated testing: Next, we conduct automated testing on the patched scripts to verify their behavior against the original version to ensure that the patches have not introduced any unintended changes to a project (*i.e.,* we check if they produce the expected outputs). For this step, we randomly selected six projects from the set of projects used in our experiment (see Section 4.3); the chosen projects for evaluation represent approximately one-third of the analyzed projects.

Human evaluation: To further validate the correctness and coverage of the code generated by *Patcher*, we conducted a human evaluation. We used the six projects selected from the previous step and sought volunteers from the Computer Science Department of Dalhousie University with prior experience developing DL models in Python using TensorFlow. Six graduate students were chosen to participate, and we assigned two projects to each evaluator, ensuring two evaluators for each project. The evaluators were provided with the original Python notebook, the converted Python script from the notebook, the method-level patched script, and the project-level patched script. They were briefed about their task using a patch template as described in Listing 1, and were informed about the purpose of the validation without indicating the authorship of *Patcher*. The evaluators were then instructed to assess the provided artifacts and document any issues related to the following criteria:

- **Correctness**: This assesses whether the generated *patch* for each method adheres to the patch template as shown in Listing 1 (*i.e.,* it accurately extracts all the argument values for a given API call). The evaluators were asked to mark each patched API call as either *Correct* or *Incorrect* based on this criteria. To evaluate, we calculate *Correctness* accuracy as the ratio of total correct patches to the total number of patches evaluated.
- **Completeness**: This evaluates whether the tool accurately identifies all TensorFlow API calls and appropriately patches all relevant calls. The evaluators were asked to provide the total number of eligible API calls (*i.e.,* TensorFlow based calls), along with the total number of calls that were missed by the *Patcher* (*i.e.,* API calls that should have been patched, but were missed by the *Patcher*). To evaluate, we calculate *Completeness* accuracy as the ratio of total patched calls to the total number of eligible calls.

The evaluators were asked to fill an anonymous Excel sheet [26] for each project containing information such as logs and relevant method calls for which they checked the patched code to verify each of the evaluation criteria. We consolidated the evaluations and checked for any differences in the evaluation for each project.

Notably, the evaluators reported high accuracy for the patched projects based on the adopted criteria. Specifically, the *Correctness* criterion achieved 100% accuracy, while the *Completeness* criterion achieved 99.3% accuracy. *Patcher* missed 1 API call out of expected 159 total calls. A detailed analysis revealed that these missed instances are API calls made via returned TensorFlow objects from user-defined functions. The *Patcher* currently operates on method calls made by the objects created and used in the same code block. Hence, when a user-defined method creates and returns an object to a code block that uses the object to invoke a method, it does not get identified by the *Patcher*. We aim to address this limitation in the future versions of the *Patcher*, by introducing type prediction of the returned objects.

3.3 Pre-measurement Stability Checks

The machine used for the experiments must be stable before executing the API calls and collecting their energy consumption as suggested by Georgiou *et al.* [32]. The introduced stability checks ensure low fluctuation in the hardware's energy consumption, thus reducing noise and ensuring

accurate measurements by conducting energy measurement experiments under approximately the same conditions every time.

We perform two kinds of stability checks as part of the FECoM framework—the *temperature check* and the *energy stability check*. GPU overheating can substantially increase power draw [65], skewing results. The temperature check ensures that the CPU and GPU temperatures are below a standard hardware-specific threshold, maintaining uniform thermal conditions. FECoM uses `lm-sensors` [51] tool to obtain the CPU temperature and `nvidia-smi` [18] to obtain GPU temperature. We follow a key guideline to run as few user processes as possible during the experiment. This guideline helps us achieve stability from a temperature and energy consumption perspective.

With energy stability check functionality, we ensure that CPU, RAM, and GPU energy observations are not fluctuating. Variability indicates outside processes are consuming significant power, introducing measurement noise. The check also accounts for overheads from static instrumentation by ensuring a steady pre-instrumentation state. Fluctuations in energy consumption indicate that other processes on the machine are consuming considerable energy, and hence, the measured energy might include considerable noise. To perform the check, we measure and record energy consumption by the three hardware components (*i.e.,* CPU, RAM, and GPU), periodically. We load the most recent 20 energy data observations for the components. Then, we determine stability by comparing the coefficient of variation ($\frac{\sigma}{\mu}$) [13] of the data points, to the stable state coefficient of variation, for each component. We calculate the stable state coefficient one time in no-load condition *i.e.,* by running only the energy measurement scripts without any other processes on the experiment machine for approximately 10 minutes. We calculate the coefficient of variation by dividing the mean by the standard deviation. If the coefficient of variation of the last 20 observations is smaller than or equal to the stable state coefficient of variation, the machine is deemed to be in a stable state. If both stable temperature and stable energy are achieved, the machine is ready to execute the experiments.

3.4 Energy Measurement Module

The energy measurement module [23] executes the selected project, measures the consumed energy, and documents the observations. We execute each project (and each API call, in turn) ten times to ensure the reliability of the measurements.

Figure 2 shows a typical power consumption profile of an API call. We measure the energy consumption by the API between the start (t_s) and end (t_e) time of the call execution. Since $Energy = Power \times time$, an API's energy consumption can be interpreted as the area between the graph as shown in Figure 2 and the x-axis—or the time integral of power. Let $E_t(m)$ be the energy consumed for a given API call at time t, where $t_s \leq t < t_e$. This measurement $E_t(m)$ is then adjusted by subtracting the stable mean energy consumption of the system, which can be attributed to background processes, resulting in $E'_t(m)$. $E'_t(m)$ is equivalent to the area between the mean stable power and the graph. We measure $E'_t(m)$ periodically for different values of t and add them to obtain $E(m)$ *i.e.,* $E(m) = \sum_{t=t_s}^{t_e} E'_t(m)$. The resulting $E(m)$ value is further averaged over 10 repetitions of the same experiment to obtain the mean energy consumption $\overline{E}(m)$. Throughout this paper, we will refer to $\overline{E}(m)$ when discussing net energy consumption in our experiments.

We store the raw measurements in JSON format, ensuring their accessibility and reproducibility via our replication package. The collected energy consumption data include energy consumption by CPU, RAM, and GPU. Additionally, we record timestamps, experiment settings such as wait time if the machine is unstable, wait time after API call execution, stable state power consumption, GPU and CPU max allowed temperature, and sizes of arguments passed to the API under measurement.

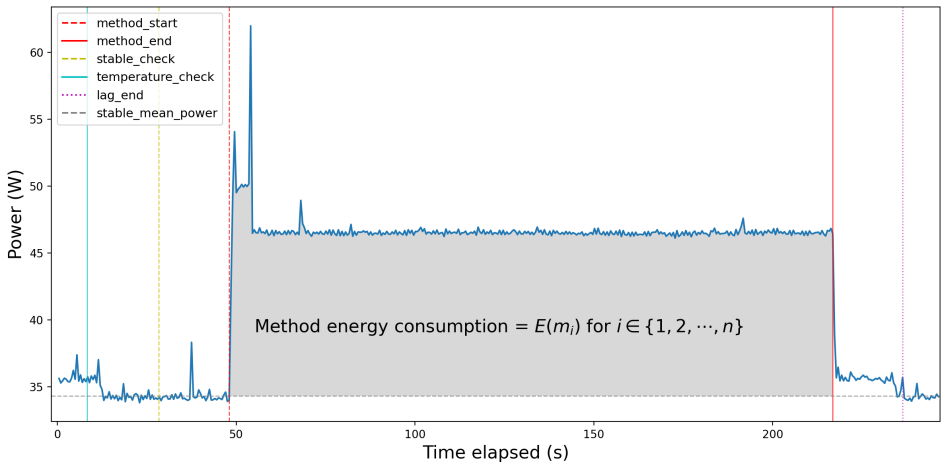


Fig. 2. CPU power over time for `models.Sequential.fit` from `images/cnn`.

3.5 Tools

We provide below a brief description of the tools that we used for energy consumption and temperature monitoring.

Intel's Running Average Power Limit (RAPL): is an interface that allows applications to monitor and control the power consumption of various components, such as the CPU and memory, within Intel processors. RAPL works by measuring the power consumption of the processor at regular intervals of approximately 1 ms [42] and reporting this information. It measures energy consumption of

- Package (PKG): all CPU components, such as cores, integrated graphics, caches and memory controllers.
- Core: all the CPU cores.
- Uncore: all caches, integrated graphics and memory controllers.
- DRAM: random access memory RAM attached to the CPU's memory controller [74].

Since the PKG values include the CPU's total energy consumption, we will only discuss these in our analysis similar to other studies [74], and we will refer to them as CPU energy consumption.

Perf: is a command-line tool for collecting performance statistics from Linux systems. Data relating to energy consumption is collected using Perf's energy event—a wrapper around Intel's RAPL. Specifically, the `perf stat` command is used to gather and report real-time performance counter statistics from running a command. Though `perf stat` supports reporting statistics at a maximum 1 ms frequency, a high overhead could result at intervals lower than 100 ms [53].

NVIDIA's System Management Interface (`nvidia-smi`) [18]: is a command-line utility that allows monitoring and controlling the performance of NVIDIA GPUs. It provides detailed real-time status of the GPU, including its power draw and temperature.

lm-sensors [51]: is a Linux software tool package that enables monitoring of the hardware sensors on the CPU, which includes temperature sensors. It provides the `sensors` command-line interface for retrieving sensor data.

3.6 Replication Package

The replication package for FECoM is available on GitHub [20]. It contains all the necessary files and instructions to replicate the experiments and results this paper presents.

4 EXPERIMENTAL SETUP

4.1 Research Questions

The goal of this study is to develop an approach and framework to measure energy consumption at a fine-grained level (*e.g.,* API level) to understand better the energy profile of APIs of DL frameworks so that it can be subsequently used to make their documentation energy-aware. Towards this goal, we propose the framework, FECoM (described in Section 3). We formulate the following research questions (RQs) to evaluate the proposed approach and the framework FECoM.

RQ1: To what extent is FECoM capable of measuring energy consumption at the API level?
In this RQ, we aim to measure the effectiveness of the proposed framework, FECoM. We want to ensure the correctness of the measured energy through this evaluation.

RQ2: To what extent does input data size have an effect on energy consumption?
With RQ2, we wish to investigate the relationship between the input data provided to the APIs under examination and their energy consumption. Answering this RQ will reveal the energy profile of APIs in relation to their input parameter size, which should become a critical component of energy-aware API documentation.

RQ3: What are the main challenges and considerations in developing fine-grained energy measurement tools for DL frameworks?
Verdecchia *et al.* [85] emphasized the significant scarcity of tools, for example, to measure energy consumption, in the Green Artificial Intelligence domain. Additionally, Bannour *et al.* [5] noted that the existing tools lack sensitivity to measure energy consumption at fine granularity. However, measuring energy consumption at lower granularities, such as API-level poses unique challenges compared to coarse-grained measurement. With this research question, we aim to uncover the key challenges, underlying reasons and considerations that arise when developing tools measuring energy consumption for fine-grained profiling of DL frameworks and models. Furthermore, answering this research question will provide insights and suggestions for researchers and practitioners, and facilitate the development of new tools and techniques in the field.

4.2 Experimental Design

In this section, we elaborate on the experimental design choices and corresponding rationale for answering the research questions.

4.2.1 ***RQ1***. Validating the correctness of the measured energy consumption at a fine-grained granularity is a non-trivial challenge due to the lack of existing tools or benchmarks to measure energy consumption at the fine-grained level. To overcome the challenge, we measure energy consumption both at the API level and at the project level, *i.e.,* we measure the total energy consumed by the whole project and the energy consumed by all the API and method calls belonging to a framework like TensorFlow. The energy consumed by a project is approximately the sum of energy consumed by all the methods defined, called (including library/framework methods and API calls), and executed within the project. Therefore, the sum of the energy consumed by the measured methods must be less than the total energy consumed by the project. Concretely, we model the relationship between energy consumed by a project $E(P)$ and the methods $E(m_i)$ for $i \in \{1, 2, \cdots, n\}$, in the project, in the following way.

$$E(P) \approx E(M_s) + E(M_o) = \sum_{i=1}^{k} E(m_i) + \sum_{i=k+1}^{n} E(m_i) \quad (1)$$

Where methods m_i for $i \in \{1, 2, \cdots, k\}$ are in the scope of energy measurement (*e.g.,* TensorFlow methods and API calls in the considered project code) representing $E(M_s)$ and, hence, measured by FECoM. The rest of the methods m_i for $i \in \{k+1, k+2, \cdots n\}$ represent $E(M_o)$ that falls outside of the scope of FECoM and, hence, we do not measure their energy consumption. The energy consumed by methods within scope *i.e.,* $E(M_s)$ cannot be greater than the energy consumed by the entire project. In Equation 1, if the energy consumed by out-of-the-scope methods is negligible, *i.e.,* $E(M_o) \approx 0$, we should observe $E(P) \approx E(M_s)$ if we measure the energy consumed by individual methods, correctly.

For example, in Listing 1, the energy consumption of the project $E(P)$ would include the energy consumed by the `tf.keras.Input()` API call, denoted as $E(m_1)$, as well as the energy consumed by `np.stack()`, `pd.concat()`, and `len()` methods, denoted as $E(m_2)$, $E(m_3)$, and $E(m_4)$, respectively. In this case, $E(M_s) = E(m_1)$ as the `tf.keras.Input()` API call is a TensorFlow API and hence within the scope of energy measurement, while $E(M_o) = E(m_2) + E(m_3) + E(m_4)$ as the other three methods are non-TensorFlow operations and hence outside the scope. Therefore, according to Equation 1, the total energy consumption of the project can be approximated as $E(P) \approx E(m_1) + E(m_2) + E(m_3) + E(m_4)$.

We extend the evaluation of our proposed approach by investigating the relationship between energy consumption and execution time at the API level granularity. Previous research suggests a linear relationship between energy consumption and execution time [14], or indirectly, execution frequency [54]. This relationship seems intuitive—the longer or more frequently a task is performed, the more energy is consumed. This assumption holds when the power P remains constant, as energy consumption E is given by $E = P \times t$, where t represents the time duration. However, if the power P is a function of time t, the linear relationship no longer applies. Nevertheless, it is generally expected that as execution time increases, energy consumption will also increase proportionally.

4.2.2 ***RQ2.*** The time complexity of an algorithm, $O(n)$, is a function describing an asymptotic upper bound of an algorithm's run-time t given an input of size n [80]. This implies that the execution time of an algorithm is a function of its input data size. In Section 4.2.1, we discuss that energy consumption y is the function of execution time *i.e.,* $y = f(t)$. Therefore, energy consumption y also is a function of input parameter size as given below.

$$y = f(n) \quad (2)$$

Though it is intuitive that the energy consumption of an API call depends on the input parameter size, the relationship between energy consumption and parameter size is not known. This research question utilizes our proposed method and FECoM to determine the concrete relationship between these two aspects.

In this experiment, we measure energy consumption by an API call multiple times, changing the passed parameters' data size. We determine $E_{CPU}(n)$, $E_{RAM}(n)$ and $E_{GPU}(n)$ *i.e.,* the energy consumed by CPU, RAM, and GPU, respectively, for various values of n. We vary the input data size n in increments of $\frac{1}{10}$ of the original data. The first run of a method uses $\frac{1}{10}$ of the original data, and this size is incremented by $\frac{1}{10}$ for each successive run until the 10^{th} run uses the entire original data size. We analyze the method-level energy consumption data from RQ1 and identify 10 of the most energy-hungry API calls. Similar to the RQ1 setup, we execute each selected API call 10 times, resulting in 100 observations for each call.

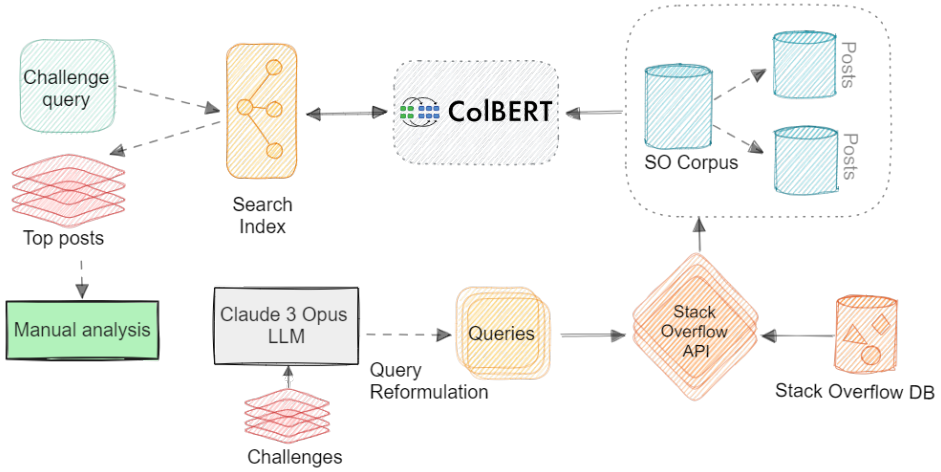


Fig. 3. RQ3 Pipeline for Stack Overflow Mining

4.2.3 ***RQ3***. This RQ explores and discusses the key considerations as well as challenges that one may face while designing and developing frameworks and tools similar to FECoM for fine-grained energy consumption measurement. To compile a comprehensive set of design considerations and challenges, we diligently documented all the error logs and issues encountered while developing the FECoM framework [21]. Initially, we carefully reviewed meeting minutes from our development team comprised of the first three authors, identifying instances where roadblocks were encountered while developing the framework. Additionally, team members individually reviewed the initial list and augmented the identified issues. They also added new issues based on their experience working on this study. We asked them to supplement the issues with additional information, including instructions to reproduce, error logs, and and error messages.

We used the Open Coding Technique [46], a qualitative data analysis method, to analyze and categorize the identified issues. This approach enabled us to systematically label and categorize the collected information, identify emerging themes and patterns, and construct a conceptual framework from the raw issues. The process involved the following key steps:

(1) *Coding process:* The initial step involved thoroughly reviewing the collected initial issues and deconstructing them into smaller segments for close examination. We analyzed these segments to identify relationships, similarities, and dissimilarities. We assigned appropriate codes to each segment to facilitate further analysis, enabling us to identify and categorize them systematically for subsequent analysis. This process was conducted in isolation for each of the team members to eliminate bias in the coding process.

(2) *Iterative process:* Open coding is an iterative process that involves revisiting the segments and individual items multiple times. After an initial round of coding, we searched for new concepts and patterns by continually refining our coding scheme until we reached data saturation.

(3) *Emergent coding:* With emergent coding, codes and categories emerged directly from the data itself, allowing themes and patterns to be identified inductively. This approach prevented us from imposing preconceived notions and theories, ensuring a data-driven analysis.

(4) *Constant comparison:* We continuously compared new data with existing codes and categories throughout the coding process to ensure consistency and identify variations or similarities.

After completing the process, we obtained issues grouped into multiple categories and subcategories.

To systematically identify mitigation strategies for the identified challenges, we followed a multi-step approach leveraging both manual analysis and automated techniques. Figure 3 summarizes the approach; we elaborate on each key step below.

(1) **Keyword extraction:** We treated each documented challenge as an initial query and used the state-of-the-art Claude-3 Opus Language Model (LLM) API [4] to extract top keyword queries that are semantically similar to the initial query. We chose Claude-3 for its strong performance in natural language understanding and generation tasks. However, depending on the availability and specific requirements, it can be replaced with any other similarly capable LLM. This step helps address the potential Lexical Chasm problem [6], which refers to the mismatch between developers' vocabulary and terminology when describing their challenges and the actual content and language used in relevant documentation or Q&A forums such as Stack Overflow. By extracting semantically similar keyword queries, we bridge this lexical gap and improve the chances of finding relevant information in subsequent steps [50].
(2) **Corpus construction:** We utilized the extracted keyword queries to search for relevant Stack Overflow posts using the Stack Overflow API provided through the Stack Exchange platform [19]. This process yielded a corpus of total 1, 367, 422 potentially relevant posts for all identified challenges.
(3) **Corpus filtering:** To filter out irrelevant posts, we created an Information Retrieval based search pipeline. This pipeline combines the challenge descriptions with the Stack Overflow posts using *ColBERT* [47], a state-of-the-art retrieval model. We constructed a search index of the Stack Overflow posts using ColBERT, enabling efficient querying and retrieval. Using the challenge descriptions as search queries, we queried the index and extracted the top 50 most relevant posts for each challenge. The search index and corpus used in this study are available as an artifact [72].
(4) **Manual analysis:** Finally, To ensure the comprehensiveness and reliability of the filtered posts, the first author manually analyzed the filtered corpus of Stack Overflow posts to identify the ones discussing challenges and potential mitigation strategies relevant to our context. This involved reviewing multiple posts for each challenge, extracting relevant mitigation strategies, and summarizing them. We provide the mapping between the specific posts used for this analysis and the corresponding identified challenge to facilitate traceability in an Excel sheet within our replication package [25]. Furthermore, we provide the created index and the post dataset as part of the replication package [72], empowering users to search for any desired number of posts based on their specific use cases. Users also have the flexibility to expand the trained index by incorporating additional posts relevant to their requirements.

After completing this process, we obtained a categorized list of challenges encountered while developing tools for fine-grained energy measurement and their mitigation strategies. The identified challenges provide valuable insights into the key considerations and potential pitfalls when designing and developing such tools. This RQ focuses on identifying and understanding the challenges themselves rather than prescribing specific solutions, as the effectiveness of the mitigation strategies may vary depending on the context.

4.3 Repository Selection

The following criteria were used to select a DL project repository for evaluating the RQs.

(1) It should be publicly available on GitHub and popular (*i.e.,* having at least 5, 000 stars).
(2) It should include a good variety of TensorFlow-2-based projects across various DL domains and should be actively maintained.

(3) It should be easily reproducible.

Based on these criteria, we chose the DL tutorials repository from the official TensorFlow documentation [83] (commit: e7f81c2) for our experiments. This repository is publicly available on GitHub, with over 5.7 thousand stars and 5.1 thousand forks, and offers a wide variety of DL tasks, including computer vision, natural language processing, and audio processing, each presented as a self-contained Jupyter notebook. These tutorials extensively utilize TensorFlow's essential APIs. Moreover, they are designed for easy reproducibility, as each tutorial performs all the steps of the DL pipeline (*i.e.,* data loading, pre-processing, training, testing) within the notebook without any additional dependencies. Additionally, the tutorials are continually updated to work with the latest TensorFlow versions, maintained by a team of more than 800 contributors.

4.4 Experimental Environment

All experiments were conducted on a Ubuntu 22.04 machine equipped with an Intel(R) Xeon(R) Gold 5317 CPU (24 logical cores, 3.00 GHz), and 125 GB of main memory. For GPU-accelerated computations, the machine incorporates an NVIDIA GeForce RTX 3070 Ti GPU [60] with 8 GB GDDR6X MEMORY. The GPU exhibits an idle power of 18 Watts and maximum power consumption of 290 Watts. To ensure consistent and reliable results, our experimental setup utilizes Python 3.9 alongside TensorFlow 2.11.0. This combination requires NVIDIA CUDA 11.2 and CUDNN 8.1.0 to leverage the full capabilities of the GPU and optimize DL computations.

4.5 Settings

4.5.1 Sampling interval. The frequency at which energy measurement samples are captured and retrieved is an important factor in measuring energy consumption. The frequency of the sampling interval affects the precision and resolution of the measurements taken. A high-frequency sampling interval can provide more precise and detailed measurements. However, it may also require more processing power and resources and generate a larger data volume (leading to more overheads) [53]. However, using a lower frequency sampling interval can also result in situations where energy consumption readings at API granularity cannot be captured precisely. If a API's execution time is shorter than the sampling interval, energy consumption cannot be measured effectively. Thus, this trade-off needs consideration in selecting a sampling interval frequency.

To determine an appropriate sampling interval, we conducted experiments with FECoM's baseline that would denote the overhead using a range of sampling frequencies from 1ms to 1000ms and analyzed the coefficient of variation (CV) of the energy measurements. CV measures the variability in the measurements relative to the mean [13]. Figure 4 presents the CV values for CPU, RAM, and GPU energy measurements across different sampling intervals.

As the figure illustrates, the CV values generally decrease as the sampling interval increases while stagnating around and after 500ms, indicating that higher sampling intervals result in more stable and consistent measurements for the considered APIs. Very low sampling intervals, such as 1 ms and 10 ms, exhibit significantly higher CVs compared to the chosen 500 ms interval, suggesting that very high sampling frequencies introduce more noise, variability and hence overhead in the measurements, which can be detrimental to the accuracy and reliability of the energy consumption analysis. The 500 ms sampling interval achieves a good balance between measurement stability (low CV) and granularity. While even higher intervals like 1000 ms show slightly lower CVs, they may miss capturing the energy consumption of shorter API calls. The 500 ms interval strikes a reasonable compromise, ensuring stable measurements while still capturing the energy profiles of the most relevant API calls. This interval aligns with the range of frequencies used in software energy measurement studies, which typically span from milliseconds to seconds, as shown in Table 1.

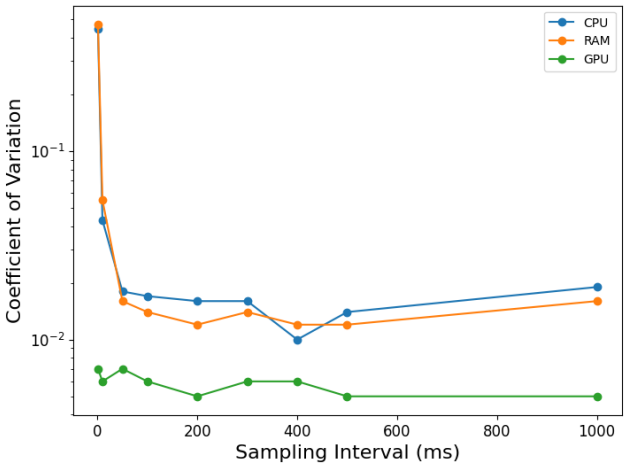


Fig. 4. Coefficients of Variation across sampling intervals

Furthermore, users of FECoM have the flexibility to customize the sampling interval in the FECoM configuration based on their specific experimental requirements. If finer-grained measurements are desired for particular use cases, the sampling interval can be adjusted accordingly, taking into account the potential trade-offs in terms of overhead and data volume.

4.5.2 Machine stability. As discussed in Section 3.3, maintaining the machine's temperature and energy stability is crucial to ensure accurate and reliable energy consumption measurements. We have set conservative temperature thresholds of 55°C for the CPU and 40°C for the GPU, well below the maximum allowed temperatures of 84°C for the Intel(R) Xeon(R) Gold 5317 CPU [43] and 93°C for the NVIDIA GeForce RTX 3070 GPU [60] as specified in the product documentation. We adopt best practices from the literature to achieve and maintain stable conditions during measurements. For the GPU `nvidia-smi`, enabling persistence mode is crucial as it optimizes GPU performance by keeping it in a fixed state and CUDA libraries ready for immediate use, minimizing voltage and frequency changes [10, 66]. Similarly, setting the CPU power policy to performance mode ensures it operates at maximum frequency [81].

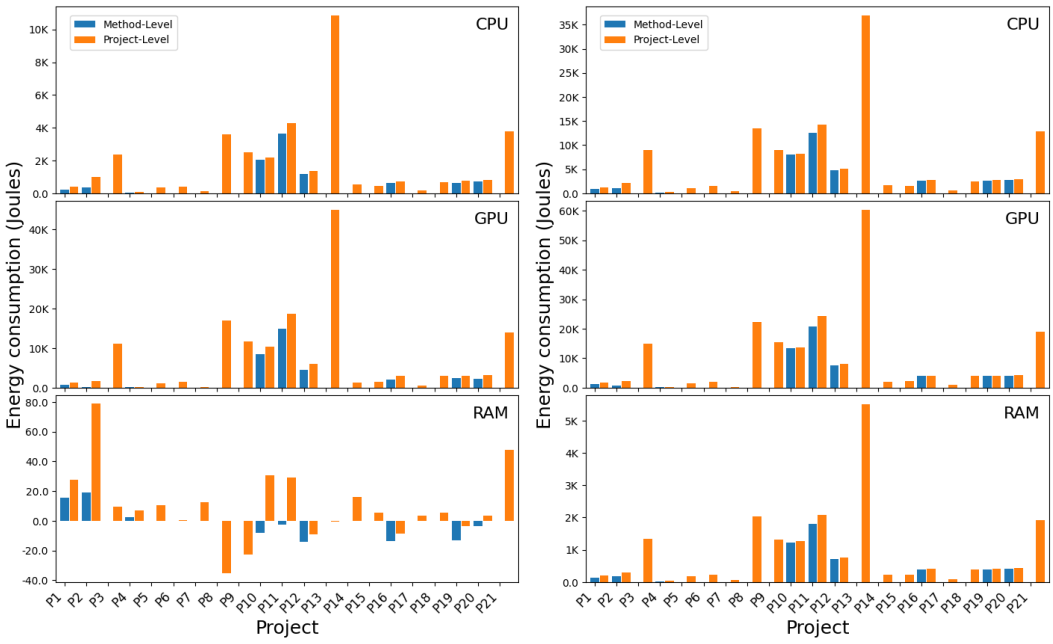
To enhance measurement accuracy, it is essential to minimize background processes on the machine related to energy measurement. Stopping unnecessary background processes [32] ensures that only the relevant processes run during the measurement cycle. Additionally, the energy measurement cycle for each API call should only initiate when the machine is in a "stable condition" [32]. This is ensured via the energy stability check discussed in Section 3.3. After each call execution, the machine should remain idle for a brief period to avoid tail power states [32], ensuring the accuracy and consistency of the energy consumption measurements.

5 RESULTS

In this section, we present our experimental observations for the considered research questions.

5.1 RQ1. FECoM Effectiveness

To answer RQ1, we measure energy consumption both at the API and the project level. As we discuss in Section 4.2.1, if the sum of energy consumed by the measured APIs is greater than the energy consumed by the entire project, then the proposed approach is falling short of measuring



(a) Net energy consumption

(b) Total energy consumption

Fig. 5. RQ1. Net and total energy consumption (CPU, GPU, and RAM) at project and method levels.

energy consumption at the fine-grained granularity. We use approximate values for the comparison because energy consumption tools have limited precision [15, 31]. As we discuss in Section 4.5.1, we use a sampling interval of 500 ms for the `perf` and `nvidia-smi` tools. It implies that the energy consumption of any API with a run time below 500 ms is excluded from the sample of considered calls, further reducing the sum of method-level energy consumption values.

Figure 5a and Figure 5b show the net and total energy consumed by individual projects, respectively, and all methods in the scope within individual projects for each hardware device (*i.e.,* CPU, GPU, and RAM). Figure 5a shows that $E(P) > E(M_s)$ for all analyzed projects for each hardware device considered in our study. We can observe varying degree of difference between $E(P)$ and $E(M_s)$ for different projects. Such a variance stems from different amounts of energy consumed by methods not in the scope, *i.e.,* $E(M_o)$. For instance, the project `images/cnn` has 15 TensorFlow methods in scope that consume 8,513.61 Joules energy within the GPU; the rest of the energy 1,801.24 Joules energy is consumed by out-of-scope 14 methods. Project `keras/overfit_and_underfit` presents an intriguing case where we observe $E(P) >> E(M_s)$. This difference can be attributed to the fact that the project imports TensorFlow library in their source code, but it majorly utilizes a different library, other than TensorFlow for ML tasks. Consequently, while the energy consumption associated with this external library is reflected in the project-level energy measurement, it remains unaccounted for in the method-level energy analysis. Given that FECoM targets TensorFlow methods, energy consumed by functions of other libraries is not accounted for in $E(M_s)$.

We perform statistical tests to determine the significance of the observed differences in energy consumption between method-level and project-level measurements. To assess the normality of the data, we conducted the Shapiro-Wilk [75] test at both the method and project levels with $\alpha = 0.05$. The p-values for CPU were $1.0e-04$ and $1.33e-05$, for GPU were $3.25e-05$ and $1.80e-05$, and for RAM

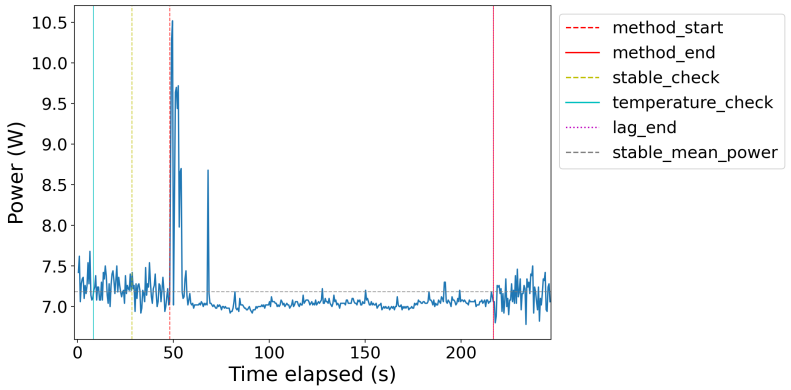


Fig. 6. RQ1. RAM power over time for `models.Sequential.fit` in `images/cnn`.

were 0.001 and 0.038, respectively for the method and project levels. The results of the Shapiro-Wilk test indicate that the data for both the method and project levels are not normally distributed ($p - values < \alpha$). As the data did not meet the normality assumption, we opted for non-parametric tests. We employed the Wilcoxon signed-rank test [92], which is suitable for paired samples, to assess whether the sum of method-level energy consumption is less than the project-level energy consumption for each energy type. We performed the Wilcoxon test using the *Scipy* [73] library with the alternative hypothesis set to "greater" that determines whether the distribution underlying the difference between paired samples (project_level and sum of method_level) is stochastically greater than a distribution symmetric about zero. Based on the results of the one-tailed Wilcoxon signed-rank test, we can conclude that there is a statistically significant difference between the method-level and project-level energy consumption for all energy types (CPU, GPU, and RAM). The p-values for the Wilcoxon signed-rank tests are extremely small($<< \alpha = 0.05$), with values ranging from $1.53e-05$ to $4.58e-05$, thus supporting the alternative hypothesis that the sum of method-level energy for a project is less than the project-level energy consumption. The above analysis concludes that the FECoM measures energy consumption at the API granularity correctly.

Figure 5a shows another interesting observation *i.e.,* the presence of negative net RAM energy consumption for some projects. Figure 6 provides an explanation of the observation. The figure shows a sudden spike in RAM power consumption at the beginning of API call execution, followed by a settling down below the mean stable power. Before executing a function on the GPU, the input data must be copied from RAM (CPU memory) to GPU memory [67]. These extensive data copying operations could explain the spike in RAM power draws at the start of execution. This spike, followed by a lower RAM power, is likely associated with TensorFlow methods utilizing GPU kernels, which utilize GPU memory VRAM instead of system RAM [82]. Comparing the numerical values of $E_{CPU}(n)$ with those of $E_{GPU}(n)$ in Figure 5a confirms that GPU energy consumption is approximately five times higher than CPU energy consumption across all experiments. After the initial data copying, the CPU and RAM enter an idle state, requiring less energy than the stable state. Consequently, the area below the mean stable power surpasses the area above it, resulting in negative average energy consumption, as illustrated in Figure 5a.

We extend the evaluation of our proposed approach by investigating the relationship between energy consumption and execution time at the method-level granularity. To validate the linear relationship between energy consumption and mean execution time, we compute the Pearson correlation coefficient. The obtained correlation coefficients ρ for GPU, CPU, and RAM are 0.99

(p-value = $9.34e-27$), 0.99 (p-value = $6.01e-39$), and −0.84 (p-value = $3.10e-08$), respectively. These values indicate that execution time for GPU and CPU have a strong positive relationship with energy consumption, while RAM exhibits a strong negative relationship. These observations further confirm the effectiveness of the proposed approach.

FECoM enables new insights into energy consumption patterns in real-world deep learning code. Consider as an example the TensorFlow program [28] for image classification as shown in Listing 2; this program loads data, trains a model, evaluates it, and performs inference. Using FECoM, we can break down the total energy usage and attribute consumption to individual APIs. The data loading operation via `fashion_mnist.load_data()` is quite efficient, consuming only 1J. In contrast, `model.fit()` for training is identified as an energy hotspot, drawing significant power at 4400J. Evaluation and inference operations exhibit more moderate consumption of 29J and 3J.

Summing the method-level measurements, the TensorFlow APIs account for about 4500J energy. However, FECoM's project-level view shows the overall program consumes 5990J. This additional 1490J can be ascribed to non-TensorFlow operations such as file I/O and data pre-processing. By supporting fine-grained profiling, FECoM reveals patterns that arise from contributions of specific API calls versus other program activities. Developers can use these insights to target optimization efforts on costly operations such as `fit()`. FECoM enables drilling down into energy consumption patterns within real DL code, empowering developers to write greener, more efficient AI applications.

```
import tensorflow as tf
....
# Energy efficient loading data : 1J
train_images, train_labels = fashion_mnist.load_data()
....
# Energy hotspot - fit consumes maximum energy: 4400J
model.fit(train_images, train_labels, epochs=10)
....
# Evaluation consumes moderate energy : 29J
test_loss, test_acc = model.evaluate(test_images, test_labels)
....
# Low energy inference: 3J
predictions = probability_model.predict(test_images)
....
# Total API-level energy consumption: 4500J
# Project-level energy consumption: 5990J
```

Listing 2. Example of a TensorFlow code snippet annotated with energy usage

Summary of RQ1: The results provide strong evidence of the effectiveness of FECoM energy consumption measurements at a fine granularity. FECoM's effective API-level energy measurements provide developers with valuable insights that can be leveraged to create more energy-efficient DL pipelines. By identifying and optimizing energy hotspots, developers can contribute to the development of greener and more sustainable AI applications.

5.2 RQ2. Effect of Input Data Size on Energy Consumption

To answer RQ2, we executed API calls with varying input data sizes. The results are presented in Figure 7. The scatter plot illustrates the Net energy consumption in Joules against the total parameter size in megabytes (MB). The figure reveals a linear relationship between input data size and energy consumption for both the CPU and GPU, indicating that $E_{CPU}(n)$ and $E_{GPU}(n)$ are increasing linear functions for this API call. This confirms the hypothesis for RQ2. However, when considering RAM energy consumption as a function of input data size, instead of an *increasing* linear function, $E_{RAM}(n)$ appears to remain *constant*.

To validate the linear relationship between energy consumption and input data size, we computed the Pearson correlation coefficient. The obtained correlation coefficients ρ for GPU, CPU, and RAM are 0.89 (p-value = $4.2e-32$), 0.88 (p-value = $3.12e-30$), and 0.19 (p-value = 0.08), respectively. These values indicate that input data size for GPU and CPU have a strong positive relationship with energy consumption, while RAM exhibits a low positive relationship. This means that as the data size increases, CPU and GPU energy consumption tends to increase as well. The extremely small p-values for CPU, and GPU signify the high statistical significance of these correlations. However, the low correlation exhibited by RAM is not statistically significant. The effect sizes, representing the magnitude of the Pearson correlation coefficients, are substantially large for the GPU and CPU, indicating a significant relationship between their energy consumption and input data size.

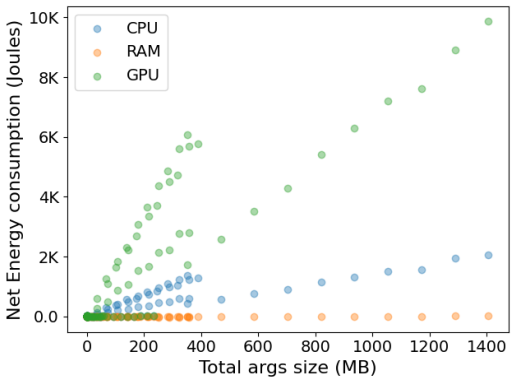


Fig. 7. RQ2. Net energy consumption Vs. method call input size.

5.2.1 Compute-intensive vs memory-intensive APIs. To provide a more nuanced analysis, we classified the APIs based on their computational resource usage into compute-intensive or memory-intensive categories, as well as their lightweight counterparts. To classify the APIs, we conducted an analysis of resource utilization patterns based on metrics runtime and data size for all the APIs in our scope. APIs with a runtime greater than 1 second were classified as compute-intensive, while those with a data size greater than 1 MB were classified as memory-intensive.

Compute-intensive methods: Compute-intensive APIs (*i.e.,* APIs with runtime > 1 second) form 61% of the total APIs. Figure 8 shows the net energy consumption for APIs against their execution runtime. We compute the correlation coefficient separately for compute-intensive methods and the rest of the methods. For compute-intensive methods, input data size has a strong positive correlation with CPU ($\rho = 0.86$, p-value = $2.00e-06$) and GPU ($\rho = 0.90$, p-value = $1.03e-07$) energy consumption, with large effect sizes and statistical significance. This suggests that as the input data

size increases, the energy consumption of CPU and GPU also increases significantly for compute-intensive methods. However, RAM energy consumption shows a moderate negative correlation ($\rho = -0.47$, p-value = $4.17e-02$) with input data size, indicating that RAM energy consumption tends to decrease slightly as input data size increases for these methods. For light-weight APIs (runtime <= 1 second), the correlations between input data size and energy consumption are not statistically significant, with small effect sizes.

Memory-intensive methods: Memory-intensive APIs (*i.e.,* APIs with argument size > 1 MB) form 23% of the total APIs. The memory-intensive APIs show a strong positive correlation between input data size and CPU ($\rho = 0.92$, p-value = $3.00e-03$) and GPU ($\rho = 0.94$, p-value = $1.70e-03$) energy consumption, with large effect sizes and statistical significance. This suggests that large data size leads to higher energy consumption for these APIs, similar to the compute-intensive APIs. However, the correlation between input data size and RAM energy consumption for memory-intensive APIs is not statistically significant. Similarly, for lightweight APIs from memory perspective, the correlations between input data size and energy consumption for CPU, GPU, and RAM are weak and not statistically significant, with small effect sizes(CPU: $\rho = -0.06$, p-value = $7.96e-01$; GPU: $\rho = -0.06$, p-value = $7.81e-01$; RAM: $\rho = 0.16$, p-value = $4.53e-01$).

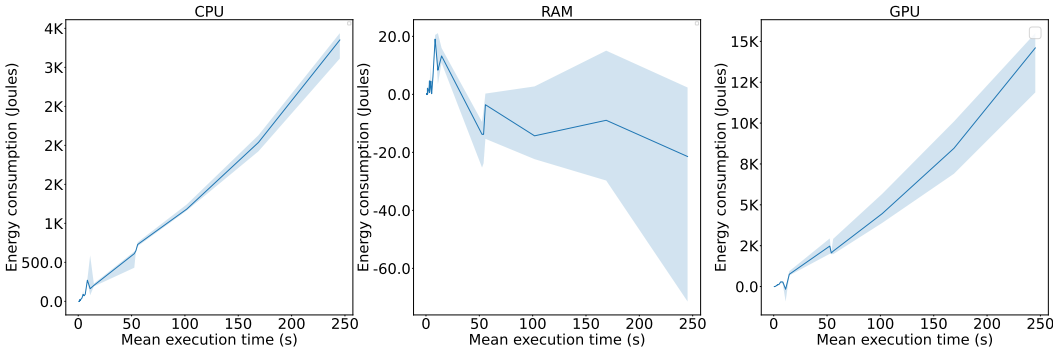


Fig. 8. RQ2. Net energy consumption Vs. execution time.

5.2.2 RAM energy consumption. The observed RAM energy consumption pattern, where $E_{RAM}(n)$ does not increase with input data size, is consistent across all the APIs analyzed in this study, including both memory-intensive and compute-intensive APIs. This can be attributed to the deep learning context of the projects considered in this study. Deep learning tasks are traditionally compute-intensive and require GPU acceleration. GPUs have their own dedicated memory (VRAM), and the energy consumption of VRAM is reported as part of the GPU's energy consumption. As discussed in RQ1, after the initial data transfer from RAM to VRAM, the RAM's role is minimal, leading to relatively constant energy consumption, irrespective of input data size.

Summary of RQ2: The results show that the energy consumption of CPU and GPU exhibits a very strong positive correlation with the input data size, particularly for compute-intensive and memory-intensive APIs. However, RAM energy consumption remains relatively constant, irrespective of input data size, due to the deep learning context where GPU memory (VRAM) plays a more significant role.

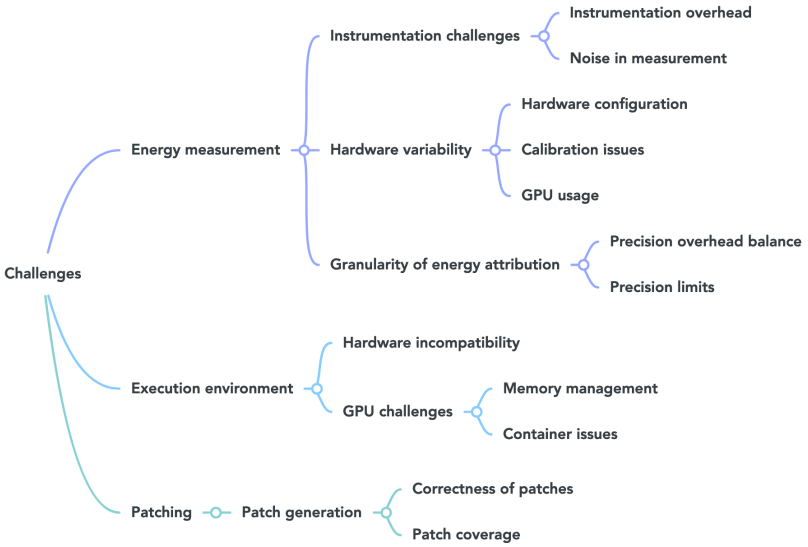


Fig. 9. RQ3. Challenges in fine-grained energy measurement.

5.3 RQ3. Key considerations in Fine-grained Energy Measurement

This research question aims to support developers and researchers working in this field by elaborating on the issues, considerations, challenges one may encounter while developing a tool similar to FECoM as well as their corresponding mitigation. Figure 9 presents the issues and challenges (organized into categories/sub-categories) that we obtained by using an open coding process as discussed in the experiment design Section 4.2.3. We provide an online appendix in the replication package, with comprehensive details about the design considerations and challenges faced while developing FECoM, along with supplementary material documenting error logs and issues used in the open coding process [21]. These issues arose and were addressed inherently during FECoM's design and development. Prior to conducting formal experiments, we tested and refined FECoM extensively by evaluating its functionality on a diverse set of projects. This rigorous verification was crucial for handling the complexities involved in fine-grained energy measurement. In the rest of the section, we elaborate on each challenge found by our analysis. We also present potential mitigation strategies for these challenges curated from Stack Overflow as outlined in Section 4.2.3.

5.3.1 ***Energy measurement***. Issues that hinder effective energy measurement belong to this category.

Instrumentation challenges: These challenges relate to the implementation of the energy measurement module.

- *Instrumentation overhead*: Instrumented code has additional instructions that may account for overhead and, therefore, may impact the performance and energy consumption of the code.
  Mitigation: We identify the following alternatives to address the challenge.
  - Minimize instrumentation points by focusing on key methods, loops, or blocks that significantly impact energy usage.
  - Use lightweight instrumentation techniques such as binary instrumentation, sampling profilers, or hardware performance counters to reduce overhead.

  - Optimize instrumentation logic by avoiding expensive operations, memory allocations, and locking within instrumented regions.
  - Quantify and subtract instrumentation overhead from energy measurements to get accurate consumption of the target code.
  - Avoid instrumentation skew by being aware of potential changes in code behavior due to heavy instrumentation.
- *Noise in measurement*: Background processes running on the machine during energy measurement introduce noise and overheads, affecting the accuracy of measured energy.
  Mitigation: To mitigate this issue, we suggest the following approaches.
  - Run code on an idle system with minimal background processes to reduce noise from unrelated activity.
  - Take a baseline energy measurement of the idle system and subtract it from the code measurements to isolate the code's energy consumption.
  - Use replicate systems to measure baseline energy simultaneously and cancel out correlated noise.
  - Characterize background noise patterns using statistical techniques like auto-correlation and apply filtering to remove noise [55].
  - Take multiple measurements and use robust statistical summaries like the median to mitigate occasional noise spikes.

Hardware variability: Issues introduced by the heterogeneity in hardware and configurations.

- *Hardware configuration*: Different hardware configurations can lead to variations in power draw and hence energy consumption values for the same project.
  Mitigation: To achieve accurate energy measurements,
  - Use standardized hardware of the same make, model, and configuration of CPU, GPU, RAM, storage, power supply, etc., for all energy measurements.
  - Ensure key hardware components are performing consistently by running benchmarks and replacing underperforming outliers.
  - Thoroughly document the exact hardware configuration, including component specs, firmware versions, and custom settings.
  - Ensure a temperature-controlled environment and document ambient temperature.
  - Disable power management features such as dynamic voltage/frequency scaling to maintain consistent behavior.
  - Take multiple measurements to characterize the expected range of variation for a given configuration.
  - Avoid concurrent workloads on the measured hardware to minimize interference.
- *Calibration issues*: Energy measurement tools require calibration to account for hardware variations.
  Mitigation: To mitigate calibration issues stemming from hardware variations, consider the following approaches:
  - Use hardware performance counters such as perf that provide a consistent interface for performance measurements across systems.
  - Employ techniques such as repeating short tests multiple times and keeping the best times to calibrate timings while accounting for hardware factors.
  - For GPUs, reduce kernel execution time during calibration by using smaller datasets or multiple kernel launches to avoid performance counter overflows.
  - Store calibrated values for stable energy consumption, maximum allowed temperatures, and wait times specific to the hardware configuration for reuse.

  - Understand the expected range of variation for your hardware and incorporate that knowledge into interpreting energy measurements and making decisions.
- GPU *usage*: The selected subject systems must utilize GPU in an optimized manner. Failing to ensure this challenge may introduce incorrect and inconsistent energy data collection.
  Mitigation: To mitigate this issue, we suggest the following approaches.
  - Monitor GPU utilization using tools such as nvidia-smi to get accurate metrics such as Volatile GPU-Util.
  - Ensure the workload is large enough to benefit from GPU acceleration, focusing on datasets where parallelization outweighs data transfer overhead.
  - Optimize kernel design and execution parameters based on the specific GPU architecture, experimenting with different configurations for optimal performance.
  - Use GPU-optimized libraries and functions when available, such as torch.linalg.solve in PyTorch.
  - Be aware of potential throttling issues, ensuring the GPU runs within its power limits and the server provides adequate cooling.
  - Profile and optimize I/O bottlenecks, such as loading tensors from the CPU, to maximize GPU utilization.
  - Monitor GPU memory usage and ensure the application's resource requirements fit within the available GPU memory, being mindful of limitations when running multiple contexts or applications concurrently.

Granularity of energy attribution: Issues related to precision limit, and to the required balance between precision of energy consumption and associated overheads.

- *Precision limits*: Existing software tools, due to Intel RAPL limitation, do not permit energy measurements at intervals smaller than 1ms.
  Mitigation: Following strategies can be considered to mitigate this issue.
  - Be aware of the limited granularity of existing APIs such as GetSystemTime() on Windows, which may have actual precision limitations despite reporting millisecond precision.
  - Understand that instrumentation and tracing memory accesses introduces overhead that limits achievable granularity. Target reasonable sampling rates based on these constraints.
  - Utilize newer hardware capabilities for finer-grained measurement where available, such as Intel Processor Trace on recent Xeon processors, while being aware of their limitations.
  - Set appropriate expectations that perfect nanosecond-level measurements are not feasible in software. Be pragmatic about the level of precision truly needed for the specific use case.
- *Precision overhead balance*: Observing energy consumption at a high frequency improves the precision of observed data; however, such high frequencies also introduce computation overhead that may introduce noise.
  Mitigation: To mitigate this issue, one may consider the following aspects.
  - Assume a noise level of around 5% when using GPU power sensors for comparison purposes, and consider averaging measurements from multiple GPUs of the same type to account for manufacturing tolerances.
  - Use oversampling with increased frequency or a higher-resolution ADC to improve signal stability and precision while verifying signal quality using an oscilloscope.
  - Employ noise-reducing algorithms, such as low-pass filters, to smooth fluctuations in the data and minimize the impact of noise on precision.
  - Adjust the sampling interval based on specific application requirements, ensuring it is high enough to minimize overhead but low enough to provide sufficient accuracy.

- Increase the number of samples used for analysis by adjusting the frame size to improve frequency resolution and precision, keeping in mind the limitations imposed by the length of the data sample.
- Consider the trade-off between latency and complexity when choosing between in-memory databases and shared-memory IPC for high-frequency data communication, weighing the advantages of data persistence and service failure recovery against potential latency and overhead.

5.3.2 ***Patching.*** Issues related to source code instrumentation.

Patch generation: This category summarizes the issues and considerations related to patch generation.

- *Correctness of patches*: Each identified patch location (in our case, each TensorFlow API) must be correctly patched to record the correct energy consumption of the API and not introduce new syntactic or semantic issues.
  Mitigation: Consider the following options to ensure the correctness of the generated patches.
  - Follow proper format and structure for patches as specified in relevant standards (*e.g.,* RFC 5789 for HTTP PATCH method).
  - Use the correct apiVersion for the resource kind being patched when applying patches with tools such as Kustomize [77].
  - Consider using a robust templating engine such as Helm for more complex templating of patches, combined with Kustomize for resource management, specific config via patches, and overlays.
  - Manually validate generated patches by reviewing the *diff* between the original and patched code.
  - Write automated tests that apply the patches and verify the patched code still compiles, runs, and produces the expected output.
  - Submit patch changes upstream to the open-source project for review, citing relevant discussions and standards to justify the approach.
  - Document the specific versions of compilers, interpreters, and dependencies the patch is compatible with.
  - Fall back to a safe emulation or unoptimized approach using conditional compilation when in doubt about patch correctness for performance optimizations.
- *Patch coverage*: Each patch location must be identified correctly to avoid missing code that is supposed to be patched and measured.
  Mitigation: Consider the following strategies to ensure the correctness of the generated patches.
  - Examine the patched code to optimize and ensure proper coverage, checking for flags between instructions, using byte-sized patches, and aligning patches on cache lines to avoid fetching multiple lines.
  - Consider using PC sampling instead of instrumenting code, periodically interrupting the thread and sampling the program counter (PC) value to provide coverage data with low overhead.
  - If patching source code, insert `covered[i]=true;` probes at the start of each basic block and let the compiler optimize them, potentially reducing probe overhead to zero for blocks inside loops.
  - After instrumenting binaries to find call sites, fix the original code, add tests, and use techniques such as wrapper classes or helper functions to make incorrect usage harder.
  - Validate instrumentation correctness by manually reviewing the patched code, ensuring all desired locations are patched and no unintended code is affected.

- Be aware that determining how to patch arbitrary code to skip lines is not portable and may break with compiler or environment changes, as offsets for patching can differ between compilers and optimization levels.

5.3.3 ***Execution environment.*** Environment-related issues that may hinder effective energy measurement.

Hardware incompatibility: Compatibility issues arise when using framework versions (*e.g.,* TensorFlow) that are not compatible with the machine's hardware or software dependencies.
Mitigation: The following strategies could be considered to mitigate the issue.

- Ensure CUDA and cuDNN versions are compatible with the TensorFlow version being used, referring to tested build configurations and downgrading or upgrading as needed.
- When running TensorFlow in containerized environments, be aware of underlying CUDA and cluster incompatibilities and select the appropriate environment based on the hardware.
- If experiencing issues with TensorFlow-GPU, install a CPU-only variant first to isolate the problem and determine if it's related to GPU incompatibilities.
- Manage dependencies carefully when using multiple frameworks with shared dependencies, considering subdividing the application into different deployables or services to manage framework requirements separately.
- Investigate kernel parameters or BIOS settings that may resolve hardware-related errors such as PCIe bus errors, such as setting `pci=nommconf` to disable message signaled interrupts (MSI).
- Verify that the installed graphics card supports the system configuration, and try removing it to see if the problem persists if experiencing issues with a specific card.

GPU challenges: Issues related to effective use of GPU fall in this category.

- *Memory management*: CUDA memory allocation errors may arise when a process cannot allocate sufficient memory during data processing on a GPU.
  Mitigation: Consider the following strategies to avoid the issue.
  - Configure TensorFlow to allocate GPU memory only as needed during runtime using the `allow_growth` option.
  - Manually handle the amount of allocated memory by streaming data and never exceeding the total memory supported by the device.
  - Check return values of CUDA memory allocation calls such as `cudaMalloc` to detect `cudaErrorMemoryAllocation` errors and deallocate memory if needed.
  - Use the `cudaMemGetInfo` function to query the free and total amount of memory available on the GPU.
  - Reduce the batch size of data loaders to process fewer samples at a time and alleviate memory pressure.
  - Select a GPU instance with more memory, such as NVIDIA Tesla P100 (16 GB) or V100 (32 GB), or use multiple GPUs for large workloads.
  - Utilize CUDA Unified Memory to oversubscribe GPU memory up to the system RAM size on Pascal and newer GPUs with CUDA 8+.
  - Change the scope and lifecycle of objects to avoid continuously creating and destroying them, pre-allocating a pool of reusable objects to amortize allocation costs and sidestep fragmentation issues.
- *Container issues*: Incompatibility of Docker containers with specific GPUs and TensorFlow versions may hinder the replication of a project.
  Mitigation: The following considerations may avoid the issue.
  - Ensure that the NVIDIA drivers and Container Toolkit are properly set up on the host system, using the "`--gpus all`" flag when starting containers.

- Be aware of compatibility issues between specific GPU models, CUDA versions, and deep learning frameworks, selecting a suitable base image from `nvidia/cuda` tags.
- If encountering issues with a custom Docker image, try using an official Docker container provided by the framework developers, such as the TensorFlow GPU images on Docker Hub.
- When building custom Docker images, avoid overwriting or messing up dependencies through redundant library installations.
- Use version-numbered images instead of the "`latest`" tag when creating containers to avoid accidentally upgrading to a newer, potentially incompatible image.
- Be aware that incompatibility issues can also arise from the base OS image used in the Dockerfile, and upgrading to a newer Docker runtime may be necessary.

5.4 Selecting an Energy Measurement Technique

Based on the findings from RQ3 and the comparative analysis in Table 1, we propose a set of criteria to help users, researchers, and developers in selecting the most suitable energy measurement technique for their specific needs. These criteria address key considerations and challenges identified in fine-grained energy measurement.

- **Measurement Granularity:** The desired level of granularity in energy measurement is a crucial factor. For system-level granularity, tools such as PowerTOP, JavaIO, or Perf may suffice. However, for more fine-grained measurements at the program, function, or API level, tools such as FECoM, CodeCarbon, FPowerTool, or Experiment Impact Tracker are more appropriate.
- **Sampling Rate:** The required sampling rate depends on the specific use case and the dynamics of the system being measured. If a sampling rate in the order of seconds is acceptable, tools such as CodeCarbon (15s) or PowerTOP can be used. For higher sampling rates in the millisecond range, FPowerTool, JavaIO, or Perf are suitable options. FECoM offers a balance with a customizable sampling rate of 500ms, which is suitable for capturing the nuances of energy consumption in deep learning APIs. Power meters such as monsoon can be used to get sampling rate in the range of microseconds.
- **Language and Framework Compatibility:** The programming language and deep learning framework being used should be considered. For Python projects utilizing popular deep learning frameworks, FECoM, CodeCarbon, and Experiment Impact Tracker are compatible choices. FPowerTool caters to Fortran/C/C++ projects, while JavaIO is designed for Java projects.
- **Hardware Support:** The target hardware architecture is another important factor. Tools such as FECoM, CodeCarbon, and Experiment Impact Tracker support measurements on CPU, GPU, and RAM. FPowerTool and JavaIO focus on CPU measurements, while PowerTOP covers both CPU and GPU. For highly accurate hardware-level measurements, specialized solutions such as Monsoon are available.
- **Stability and Automation:** If power and temperature stability are critical requirements, hardware-based solutions such as Monsoon are preferred. FECoM also offers stability in these aspects. For users requiring automated energy measurement and reporting, FECoM is currently the only tool that provides this capability out of the box.

The selection of an energy measurement technique should be a well-considered decision based on the specific characteristics and requirements of the project at hand. Users are encouraged to thoroughly assess their needs and evaluate the strengths and limitations of each technique before making a choice. Factors such as instrumentation overhead, noise mitigation, hardware variability, and execution environment compatibility should also be carefully assessed to ensure accurate and reliable energy measurements.

By considering these criteria and the specific requirements of their projects, users can make informed decisions about the most suitable energy measurement technique. For example, if fine-grained measurements at the API level, compatibility with Python and deep learning frameworks, automated reporting, and stability are priorities, FECoM would be a strong choice. On the other hand, if system-level measurements with a lower sampling rate are sufficient for a Java project, JavaIO could be a good fit.

The criteria listed above serve as a guideline to aid users in making informed decisions that align with their measurement goals and project requirements. It is important to note that while these criteria provide guidance, the selection of an energy measurement technique should be based on a comprehensive evaluation of project requirements, trade-offs, and the specific challenges and considerations discussed in RQ3.

Summary of RQ3: Our experience shows that developing a tool for DL API fine-grained energy measurement includes several challenges, such as instrumentation overhead, noise, hardware variability, and GPU usage. It is important to consider the granularity of energy attribution, striking a balance between precision and overheads. Correctness and coverage of generated patches are crucial for accurately recording API energy consumption. Moreover, the execution environment should be carefully managed to overcome hardware incompatibility and GPU-related challenges for effective energy measurement.

6 THREATS TO VALIDITY

Internal validity:
Confounds and noise. Several factors could affect a method's energy measurement through confounding. On the machine, multiple processes, including typical operating system processes, run in the background—scripts and tools for energy measurement, temperature checks, and stable state checks. All of these processes induce overheads that can skew energy consumption measurements. We employ several mitigation measures. Firstly, temperature checks are disabled before running the stable checks, and stable checks are disabled during method execution. This keeps the number of processes running during execution at a minimum. Secondly, we measure stable state energy consumption without any compute load on the machine and subtract it from the gross energy consumption to get net energy consumption.
Measurement precision. Choosing an appropriate sampling interval for measuring energy at regular intervals is an important design decision. We use a sampling interval of 500 ms. Though considering a smaller sampling interval would have given us more observations for each experiment, it also increases the overheads and noises that may lead to incorrect energy consumption values. Given that our target methods are framework APIs that typically last for minutes, if not hours, we chose a relatively low sampling interval.
Construct validity: Construct validity concerns the accuracy of the measurements and inferences using those measurements. To ensure accurate energy measurement, the *Patcher* instruments the code. The instrumentation, coupled with stability checks, ensures that the measured energy consumption is indeed consumed by the API under measurement. We validated not only the *Patcher* using automated and manual validation, but also the measured energy consumption at the API granularity.
External validity: External validity deals with the generalizability of the observed results. Given that the energy consumption is highly dependent on hardware, it may pose a threat to validity. However, to mitigate this issue, we measure gross and net energy consumption (by reducing the gross energy consumption with stable state energy consumption).

7 IMPLICATIONS

7.1 Implications for Researchers

Researchers in the field can build upon the foundation of FECoM to develop hybrid measurement approaches, especially using the findings from RQ3. Researchers can conduct empirical studies to gain insights into energy consumption patterns, revealing relationships between model architecture, hyperparameters, and energy efficiency, contributing to the development of more energy-efficient DL models and algorithms. Researchers could leverage FECoM to construct detailed energy profiles of DL models, illuminating optimization opportunities. For example, in RQ2, we showed how FECoM can be used to gain insights into the relationship between energy consumption and input data size. These insights are steps towards enriching API documentation of DL frameworks. Through extensions and new experiments with FECoM, researchers can gain a deeper understanding of energy dynamics in DL systems.

7.2 Implications for Developers

DL developers can utilize FECoM's capabilities to measure and optimize energy consumption at the granularity of framework APIs. Developers can pinpoint energy hotspots in their code down to the API call level by incorporating FECoM into their development workflows. For example, in RQ1, we observed that for the Autoencoder project [27], the same type of API call "Model.fit" consumed varying amounts of energy (5656.93 J, 7330.60 J, 133.20 J) based on the context of the call. This fine-grained profiling allows developers to make informed optimizations such as substituting inefficient APIs, streamlining data pipelines, and adopting energy-aware coding practices. FECoM ultimately enables developers to build greener, leaner DL applications by illuminating energy consumption patterns. Its ease of use and integration with popular frameworks such as TensorFlow can encourage energy awareness among the wider DL developer community.

7.3 Implications for Educators

As DL courses expand, educators can utilize FECoM to instill energy-conscious development habits among students early on. By exposing students to tools such as FECoM and its fine-grained profiling, educators highlight the significant energy demands of DL and the need for efficiency. Educators can prepare the next generation of DL developers to prioritize energy efficiency and build more sustainable AI systems by integrating FECoM in class projects and assignments to measure and optimize model energy consumption, providing hands-on experience with Green AI principles.

8 CONCLUSIONS AND FUTURE WORK

In this work, we focused on the critical aspect of energy consumption in deep learning and introduced FECoM, a fine-grained energy measurement framework. FECoM uses static instrumentation to segregate the execution of an API and to ensure machine's stability. Our experiments and evaluation have shown that the proposed framework measures consumed energy at the API granularity. Our empirical analysis investigating the influence of input parameter data size and execution time on energy consumption reveals that an API's energy consumption shows a linear relationship with input data size. Furthermore, we consolidated and categorized various considerations, challenges, and issues we faced throughout the design and development of the framework. Addressing these challenges will guide future efforts in creating fine-grained energy measurement tools. In the future, we would like to use the proposed framework to extend our empirical analysis to investigate additional aspects related to the energy profile of DL framework APIs. Exploring the role of hyper-parameters and data quality on fine-grained energy consumption will further enhance

energy profiling capabilities. Additionally, we plan to extend the application of FECoM to other popular DLframeworks like PyTorch for a comprehensive analysis of energy-efficient models.

9 ACKNOWLEDGEMENT

Saurabhsingh Rajput and Tushar Sharma are supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) through grant NSERC Discovery RGPIN/04903. Maria Kechagia and Federica Sarro are supported by the European Research Council under grant no. 741278 (EPIC).

REFERENCES

[1] Mona A Abou-Of, Ahmed H Taha, and Amr A Sedky. 2016. Trade-off between low power and energy efficiency in benchmarking. In *2016 7th International Conference on Information and Communication Systems (ICICS)*. IEEE, 322–326.
[2] Dario Amodei and Danny Hernandez. 2018. Ai and Compute. https://openai.com/blog/ai-and-compute/
[3] Lasse F Wolff Anthony, Benjamin Kanding, and Raghavendra Selvan. 2020. Carbontracker: Tracking and predicting the carbon footprint of training deep learning models. *arXiv preprint arXiv:2007.03051* (2020).
[4] Anthropic. 2024. Anthropic Claude 3 API. https://www.anthropic.com/api
[5] Nesrine Bannour, Sahar Ghannay, Aurélie Névéol, and Anne-Laure Ligozat. 2021. Evaluating the carbon footprint of NLP methods: a survey and analysis of existing tools. In *Proceedings of the Second Workshop on Simple and Efficient Natural Language Processing*. 11–21.
[6] Adam Berger, Rich Caruana, David Cohn, Dayne Freitag, and Vibhu Mittal. 2000. Bridging the lexical chasm: statistical approaches to answer-finding. In *Proceedings of the 23rd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval* (Athens, Greece) *(SIGIR '00)*. Association for Computing Machinery, New York, NY, USA, 192–199. https://doi.org/10.1145/345508.345576
[7] Luis Bote-Curiel, Sergio Muñoz-Romero, Alicia Gerrero-Curieses, and José Luis Rojo-Álvarez. 2019. Deep learning and big data in healthcare: a double review for critical beginners. *Applied Sciences* 9, 11 (2019), 2331.
[8] Déaglán Connolly Bree and Mel Ó Cinnéide. 2022. The Energy Cost of the Visitor Pattern. In *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 317–328.
[9] David Brooks, Vivek Tiwari, and Margaret Martonosi. 2000. Wattch: A framework for architectural-level power analysis and optimizations. *ACM SIGARCH Computer Architecture News* 28, 2 (2000), 83–94.
[10] Ermao Cai, Da-Cheng Juan, Dimitrios Stamoulis, and Diana Marculescu. 2017. Neuralpower: Predict and deploy energy-efficient convolutional neural networks. In *Asian Conference on Machine Learning*. PMLR, 622–637.
[11] Canonical. 2018. Ubuntu Manpage: powerstat - a tool to measure power consumption. https://manpages.ubuntu.com/manpages/bionic/man8/powerstat.8.html.
[12] CodeCarbon. 2023. CodeCarbon. https://github.com/mlco2/codecarbon
[13] Wikipedia contributors. 2024. *Coefficient of variation — Wikipedia, The Free Encyclopedia.* Retrieved May 2, 2024 from https://en.wikipedia.org/w/index.php?title=Coefficient_of_variation&oldid=1220949356
[14] Luis Corral, Anton B. Georgiev, Alberto Sillitti, and Giancarlo Succi. 2014. Can Execution Time Describe Accurately the Energy Consumption of Mobile Apps? An Experiment in Android. In *Proceedings of the 3rd International Workshop on Green and Sustainable Software* (Hyderabad, India) *(GREENS 2014)*. Association for Computing Machinery, New York, NY, USA, 31–37. https://doi.org/10.1145/2593743.2593748
[15] Luís Cruz. 2021. Tools to Measure Software Energy Consumption from your Computer. http://luiscruz.github.io/2021/07/20/measuring-energy.html. https://doi.org/10.6084/m9.figshare.19145549.v1 Blog post..
[16] William J. Dally, Stephen W. Keckler, and David B. Kirk. 2021. Evolution of the Graphics Processing Unit (GPU). *IEEE Micro* 41, 6 (2021), 42–51. https://doi.org/10.1109/MM.2021.3113475
[17] Spencer Desrochers, Chad Paradis, and Vincent M Weaver. 2016. A validation of DRAM RAPL power measurements. In *Proceedings of the Second International Symposium on Memory Systems*. 455–470.
[18] NVIDIA Developer. 2022. Nvidia System Management Interface. https://developer.nvidia.com/nvidia-system-management-interface
[19] Stack Exchange. 2024. Stack Exchange API. https://api.stackexchange.com/
[20] FECoM. 2023. FECoM: A Framework to Measure Energy Consumption of TensorFlow APIs. https://github.com/SMART-Dal/FECoM/blob/main/README.md
[21] FECoM. 2023. FECoM Challenges. https://github.com/SMART-Dal/FECoM/blob/main/replication/documents/README.md
[22] FECoM. 2023. FECoM Dataset. https://github.com/SMART-Dal/FECoM/blob/main/data/README.md

[23] FECoM. 2023. FECoM Measurement. https://github.com/SMART-Dal/FECoM/blob/main/fecom/measurement/README.md

[24] FECoM. 2023. FECoM Patcher. https://github.com/SMART-Dal/FECoM/blob/main/fecom/patching/README.md

[25] FECoM. 2023. FECoM RQ3 Stack Overflow mining. https://github.com/SMART-Dal/FECoM/tree/main/replication/rq3_analysis

[26] FECoM. 2023. FECoM Validation. https://github.com/SMART-Dal/FECoM/blob/main/replication/validation/README.md

[27] FECoM. 2023. Tensorflow Tutorial Autoencoder. https://github.com/tensorflow/docs/blob/master/site/en/tutorials/generative/autoencoder.ipynb

[28] FECoM. 2023. Tensorflow Tutorial Classification. https://github.com/tensorflow/docs/blob/master/site/en/tutorials/keras/classification.ipynb

[29] Alcides Fonseca, Rick Kazman, and Patricia Lago. 2019. A Manifesto for Energy-Aware Software. *IEEE Software* 36, 6 (2019), 79–82.

[30] Eva García-Martín, Crefeda Faviola Rodrigues, Graham Riley, and Håkan Grahn. 2019. Estimation of energy consumption in machine learning. *J. Parallel and Distrib. Comput.* 134 (2019), 75–88.

[31] Stefanos Georgiou, Maria Kechagia, Panos Louridas, and Diomidis Spinellis. 2018. What Are Your Programming Language's Energy-Delay Implications?. In *Proceedings of the 15th International Conference on Mining Software Repositories* (Gothenburg, Sweden) *(MSR '18)*. Association for Computing Machinery, New York, NY, USA, 303–313. https://doi.org/10.1145/3196398.3196414

[32] Stefanos Georgiou, Maria Kechagia, Tushar Sharma, Federica Sarro, and Ying Zou. 2022. Green AI: Do Deep Learning Frameworks Have Different Costs?. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 1082–1094. https://doi.org/10.1145/3510003.3510221

[33] Sudhanva Gurumurthi, Anand Sivasubramaniam, Mary Jane Irwin, Narayanan Vijaykrishnan, and Mahmut Kandemir. 2002. Using complete machine simulation for software power estimation: The softwatt approach. In *Proceedings Eighth International Symposium on High Performance Computer Architecture*. IEEE, 141–150.

[34] Marcus Hähnel, Björn Döbel, Marcus Völp, and Hermann Härtig. 2012. Measuring Energy Consumption for Short Code Paths Using RAPL. *SIGMETRICS Perform. Eval. Rev.* 40, 3 (jan 2012), 13–17. https://doi.org/10.1145/2425248.2425252

[35] Song Han, Huizi Mao, and William J. Dally. 2016. Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding. In *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, Yoshua Bengio and Yann LeCun (Eds.). http://arxiv.org/abs/1510.00149

[36] Shuai Hao, Ding Li, William GJ Halfond, and Ramesh Govindan. 2012. Estimating Android applications' CPU energy usage via bytecode profiling. In *2012 First international workshop on green and sustainable software (GREENS)*. IEEE, 1–7.

[37] Ammar Haydari and Yasin Yılmaz. 2020. Deep reinforcement learning for intelligent transportation systems: A survey. *IEEE Transactions on Intelligent Transportation Systems* 23, 1 (2020), 11–32.

[38] Peter Henderson, Jieru Hu, Joshua Romoff, Emma Brunskill, Dan Jurafsky, and Joelle Pineau. 2020. Towards the Systematic Reporting of the Energy and Carbon Footprints of Machine Learning. arXiv:2002.05651 [cs.CY]

[39] Peter Henderson, Jieru Hu, Joshua Romoff, Emma Brunskill, Dan Jurafsky, and Joelle Pineau. 2020. Towards the systematic reporting of the energy and carbon footprints of machine learning. *Journal of Machine Learning Research* 21, 248 (2020), 1–43.

[40] Monsoon Solutions Inc. [n. d.]. High Voltage Power Monitor. https://www.msoon.com/online-store/High-Voltage-Power-Monitor-p90002590

[41] Intel. 2019. Intel® Power Gadget. https://www.intel.com/content/www/us/en/developer/articles/tool/power-gadget.html

[42] Intel. 2020. Running Average Power Limit Energy Reporting / CVE-2020-8694 , CVE-2020-8695 / INTEL-SA-00389. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/running-average-power-limit-energy-reporting.html

[43] Intel. 2023. Intel Xeon Gold Processor. https://www.intel.com/content/www/us/en/products/sku/215272/intel-xeon-gold-5317-processor-18m-cache-3-00-ghz/specifications.html

[44] Sorin Liviu Jurj, Flavius Opritoiu, and Mircea Vladutiu. 2020. Environmentally-friendly metrics for evaluating the performance of deep learning models and systems. In *Neural Information Processing: 27th International Conference, ICONIP 2020, Bangkok, Thailand, November 23–27, 2020, Proceedings, Part III 27*. Springer, 232–244.

[45] Kashif Nizam Khan, Mikael Hirki, Tapio Niemi, Jukka K. Nurminen, and Zhonghong Ou. 2018. RAPL in Action: Experiences in Using RAPL for Power Measurements. *ACM Trans. Model. Perform. Eval. Comput. Syst.* 3, 2, Article 9 (mar 2018), 26 pages. https://doi.org/10.1145/3177754

[46] Shahedul Huq Khandkar. 2009. Open coding. *University of Calgary* 23, 2009 (2009).
[47] Omar Khattab and Matei Zaharia. 2020. ColBERT: Efficient and Effective Passage Search via Contextualized Late Interaction over BERT. arXiv:2004.12832 [cs.IR]
[48] Mark Labbe. 2021. Energy consumption of AI poses environmental problems. https://www.techtarget.com/searchenterpriseai/feature/Energy-consumption-of-AI-poses-environmental-problems
[49] Mario Linares-Vásquez, Gabriele Bavota, Carlos Bernal-Cárdenas, Rocco Oliveto, Massimiliano Di Penta, and Denys Poshyvanyk. 2014. Mining Energy-Greedy API Usage Patterns in Android Apps: An Empirical Study. In *Proceedings of the 11th Working Conference on Mining Software Repositories* (Hyderabad, India) *(MSR 2014)*. Association for Computing Machinery, New York, NY, USA, 2–11. https://doi.org/10.1145/2597073.2597085
[50] Xiaoyu Liu, Shunda Pan, Qi Zhang, Yu-Gang Jiang, and Xuanjing Huang. 2018. Generating Keyword Queries for Natural Language Queries to Alleviate Lexical Chasm Problem. In *Proceedings of the 27th ACM International Conference on Information and Knowledge Management* (Torino, Italy) *(CIKM '18)*. Association for Computing Machinery, New York, NY, USA, 1163–1172. https://doi.org/10.1145/3269206.3271727
[51] Lm-Sensors. 2023. LM-Sensors GitHub Repository. https://github.com/lm-sensors/lm-sensors
[52] Irene Manotas, Lori Pollock, and James Clause. 2014. Seeds: A software engineer's energy-optimization decision support framework. In *Proceedings of the 36th International Conference on Software Engineering*. 503–514.
[53] Linux manual page. 2023. perf-stat— Linux manual page. https://man7.org/linux/man-pages/man1/perf-stat.1.html.
[54] Andrea McIntosh, Safwat Hassan, and Abram Hindle. 2019. What can Android mobile app developers do about the energy consumption of machine learning? *Empirical Software Engineering* 24, 2 (April 2019), 562–601. https://doi.org/10.1007/s10664-018-9629-2
[55] Amberle McKee. 2023. A Data Scientist's Guide to Signal Processing. https://www.datacamp.com/tutorial/a-data-scientists-guide-to-signal-processing
[56] Abolfazl Mehbodniya, Izhar Alam, Sagar Pande, Rahul Neware, Kantilal Pitambar Rane, Mohammad Shabaz, and Mangena Venu Madhavan. 2021. Financial fraud detection in healthcare using machine learning and deep learning techniques. *Security and Communication Networks* 2021 (2021), 1–8.
[57] Microsoft. 2023. Pylance Language Server. https://github.com/microsoft/pylance-release
[58] Ibtehal Talal Nafea. 2018. Machine learning in educational technology. *Machine learning-advanced techniques and emerging applications* (2018), 175–183.
[59] Anand Nayyar, Lata Gadhavi, and Noor Zaman. 2021. Machine learning in healthcare: review, opportunities and challenges. *Machine Learning and the Internet of Medical Things in Healthcare* (2021), 23–45.
[60] NVIDIA. 2023. Nvidia Geforce RTX 3070. https://www.nvidia.com/en-us/geforce/graphics-cards/30-series/rtx-3070-3070ti/
[61] Zakaria Ournani, Romain Rouvoy, Pierre Rust, and Joel Penhoat. 2021. Evaluating The Energy Consumption of Java I/O APIs. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 1–11. https://doi.org/10.1109/ICSME52107.2021.00007
[62] Ahmet Murat Ozbayoglu, Mehmet Ugur Gudelek, and Omer Berat Sezer. 2020. Deep learning for financial applications: A survey. *Applied Soft Computing* 93 (2020), 106384.
[63] Arjun Panesar. 2019. *Machine learning and AI for healthcare*. Springer.
[64] Carlo Perrotta and Neil Selwyn. 2020. Deep learning goes to school: Toward a relational understanding of AI in education. *Learning, Media and Technology* 45, 3 (2020), 251–269.
[65] Danny C Price, Michael A Clark, Benjamin R Barsdell, Ronald Babich, and Lincoln J Greenhill. 2016. Optimizing performance-per-watt on GPUs in high performance computing: Temperature, frequency and voltage effects. *Computer Science-Research and Development* 31, 4 (2016), 185–193.
[66] Carlos Reaño, Federico Silla, Gilad Shainer, and Scot Schultz. 2015. Local and Remote GPUs Perform Similar with EDR 100G InfiniBand. In *Proceedings of the Industrial Track of the 16th International Middleware Conference* (Vancouver, BC, Canada) *(Middleware Industry '15)*. Association for Computing Machinery, New York, NY, USA, Article 4, 7 pages. https://doi.org/10.1145/2830013.2830015
[67] Princeton Research. 2023. GPU Computing. https://researchcomputing.princeton.edu/support/knowledge-base/gpu-computing
[68] Cagri Sahin, Furkan Cayci, James Clause, Fouad Kiamilev, Lori Pollock, and Kristina Winbladh. 2012. Towards power reduction through improved software design. In *2012 IEEE Energytech*. IEEE, 1–6.
[69] Cagri Sahin, Lori Pollock, and James Clause. 2014. How do code refactorings affect energy usage?. In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. 1–10.
[70] Federica Sarro. 2023. Search-Based Software Engineering in the Era of Modern Software Systems. In *Procs. of the 31st IEEE RE*. IEEE.
[71] Rajput Saurabhsingh. 2024. FECoM. https://doi.org/10.5281/zenodo.11114342
[72] Rajput Saurabhsingh. 2024. FECoM Stack Overflow Search Index. https://doi.org/10.5281/zenodo.11114352

[73] Scipy. 2023. scipy.stats.wilcoxon — SciPy v1.11.1 Manual. https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.wilcoxon.html

[74] Shriram Shanbhag and Sridhar Chimalakonda. 2022. On the Energy Consumption of Different Dataframe Processing Libraries – An Exploratory Study. https://doi.org/10.48550/ARXIV.2209.05258

[75] S. S. Shapiro and M. B. Wilk. 1965. An Analysis of Variance Test for Normality (Complete Samples). *Biometrika* 52, 3/4 (1965), 591–611. http://www.jstor.org/stable/2333709

[76] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2019. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. https://doi.org/10.48550/ARXIV.1909.08053

[77] Kubernetes SIGs. 2024. *kustomize*. Retrieved May 2, 2024 from https://github.com/kubernetes-sigs/kustomize

[78] Tajana Šimunić, Luca Benini, and Giovanni De Micheli. 1999. Cycle-accurate simulation of energy consumption in embedded systems. In *Proceedings of the 36th annual ACM/IEEE Design Automation Conference*. 867–872.

[79] Tajana Simunic, Luca Benini, Giovanni De Micheli, and Mat Hans. 2000. Source code optimization and profiling of energy consumption in embedded systems. In *Proceedings 13th International Symposium on System Synthesis*. IEEE, 193–198.

[80] Michael Sipser. 2010. *Introduction to the theory of computation* (2. ed., internat. ed., [nachdr.] ed.). Course Technology, Boston.

[81] Marek Suchanek, Milan Navratil, Don Domingo, and Laura Bailey. 2018. 4.4. CPU frequency governors Red Hat enterprise linux 6. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/performance_tuning_guide/s-cpu-cpufreq

[82] TensorFlow. 2021. GPU Memory Growth. https://www.tensorflow.org/guide/gpu#limiting_gpu_memory_growth

[83] Tensorflow. 2023. Docs/site/en/tutorials at master · tensorflow/docs. https://github.com/tensorflow/docs/tree/master/site/en/tutorials

[84] Arjan van de Ven. [n. d.]. The Linux PowerTOP Tool. https://github.com/fenrus75/powertop

[85] Roberto Verdecchia, June Sallou, and Luís Cruz. 2023. A systematic review of Green AI. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* (2023), e1507.

[86] Matthew Veres and Medhat Moussa. 2019. Deep learning for intelligent transportation systems: A survey of emerging trends. *IEEE Transactions on Intelligent transportation systems* 21, 8 (2019), 3152–3168.

[87] V Vijayalakshmi and K Venkatachalapathy. 2019. Comparison of predicting student's performance using machine learning algorithms. *International Journal of Intelligent Systems and Applications* 11, 12 (2019), 34.

[88] Shangwen Wang, Ming Wen, Bo Lin, Hongjun Wu, Yihao Qin, Deqing Zou, Xiaoguang Mao, and Hai Jin. 2020. Automated patch correctness assessment: How far are we?. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 968–980.

[89] Yuan Wang, Dongxiang Zhang, Ying Liu, Bo Dai, and Loo Hay Lee. 2019. Enhancing transportation systems via deep learning: A survey. *Transportation research part C: emerging technologies* 99 (2019), 144–163.

[90] Vince Weaver. [n. d.]. Linux support for Power Measurement Interfaces. https://web.eece.maine.edu/~vweaver/projects/rapl/rapl_support.html

[91] Guang Wei, Depei Qian, Hailong Yang, Zhongzhi Luan, and Lin Wang. 2019. FPowerTool: A Function-Level Power Profiling Tool. *IEEE Access* 7 (2019), 185710–185719. https://doi.org/10.1109/ACCESS.2019.2961507

[92] Frank Wilcoxon. 1992. Individual comparisons by ranking methods. In *Breakthroughs in Statistics: Methodology and Distribution*. Springer, 196–202.

[93] Tien-Ju Yang, Yu-Hsin Chen, and Vivienne Sze. 2017. Designing Energy-Efficient Convolutional Neural Networks Using Energy-Aware Pruning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[94] Xingzhou Zhang, Yifan Wang, and Weisong Shi. 2018. {pCAMP}: Performance Comparison of Machine Learning Packages on the Edges. In *USENIX workshop on hot topics in edge computing (HotEdge 18)*.